Developer documentation

Выпуск 5.8.8

VST Consulting

нояб. 01, 2023

Оглавление

VST Utils это небольшой фреймворк для быстрой генерации одностраничных приложений. Основная отличительная черта фреймворка VST Utils это автоматически генерируемый графический интерфейс, который формируется на основании схемы OpenAPI. Схема OpenAPI это JSON, который содержит описание моделей, использованных в REST API и информацию обо всех путях этого приложения

В документации вы можете найти информацию о быстром старте нового проекта, основанного на VST Utils, описание базовых моделей, view функций и полей, доступных в фреймворке, также вы узнаете, как можно переопределить некоторые стандартные модели, view функции и поля в вашем проекте.

1 Быстрый старт

Создать новый проект, основанный на фреймворке VST Utils, довольно просто. Мы рекомендуем создавать виртуальное окружение отдельно для каждого из ваших проектов, чтобы избежать конфликтов в системе.

Давайте учиться на примерах. Все что вам нужно сделать это запустить несколько команд. Этот мануал состоит из двух частей:

1. Описание процесса создания нового приложения и основных команд для запуска и развертывания.
2. Описание процесса создания новых сущностей в приложении.

1.1 Создание нового приложения

В этом руководстве мы создадим базовое приложение.

1. **Установка VST Utils**

   ```
   pip install vstutils
   ```

   В данном случае мы устанавливаем пакет с минимальным набором зависимостей для создания новых проектов. Однако, внутри проекта используется специальный аргумент *prod* который дополнительно устанавливает все пакеты, необходимые для работы в окружении для развертывания. Также имеется набор зависимостей для тестирования, в котором содержится все, что нужно для тестирования и анализа покрытия кода.

   Также стоит отметить дополнительно зависимости, такие как:

   - **rpc** - установка зависимостей для выполнения асинхронных задач
   - **ldap** - набор зависимостей для поддержки авторизации ldap
   - **doc** - все, что нужно для построения документации и осуществления доставки документации на запущенный сервер
   - **pil** - библиотека для корректной работы валидации изображений
   - **boto3** - дополнительный набор зависимостей для работы с S3 хранилищем вне AWS
   - **sqs** - набор зависимостей для соединения асинхронных задач с SQS очередями (может использоваться вместо **rpc**).

Вы можете комбинировать разные зависимости одновременно, чтобы собрать ваш функциональный набор в проекте. Например, для работы приложения с асинхронными задачами и медиахранилищем в MinIO вам потребуется следующая команда:

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

Чтобы установить наиболее полный набор зависимостей, вы можете использовать обычный параметр **all**.

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

2. **Создание нового проекта, основанного на VST Utils**

   Если вы впервые используете vstutils, вам нужно позаботиться о начальной настройке. А именно: вам необходимо будет автоматически сгенерировать код, создающий приложение vstutils, включая конфигурацию базы данных, специфичные опции для Django и vstutils, а также настройки, специфичные для приложения. Для создания нового проекта выполните следующую команду:

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

   Эта команда предложит указать такие параметры нового приложения, как:

   - **project name** - имя вашего нового приложения;
   - **project guiname** - имя вашего нового приложения, которое будет использоваться в GUI(веб-интерфейсе);
   - **project directory** - путь к директории, в которой будет создан проект.

   Или вы можете выполнить следующую команду, которая содержит всю необходимую информацию для создания нового проекта.

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

   Эта команда создает новый проект без подтверждения какой-либо информации.

   Эти команды создают несколько файлов в `project directory`:

   ```
   /{{app_dir}}/{{app_name}}
    ├── .coveragerc
    ├── frontend_src
    │   ├── app
    │   │   └── index
    │   ├── .editorconfig
    │   ├── .eslintrc.js
    │   └── .prettierrc
    ├── MANIFEST.in
    ├── package.json
    ├── .pep8
    ├── README.rst
    ├── requirements-test.txt
    ├── requirements.txt
    ├── setup.cfg
    ├── setup.py
    ├── {{app_name}}
    │   ├── __init__.py
    │   ├── __main__.py
    │   ├── models
   ```

(continues on next page)

(продолжение с предыдущей страницы)

```
│   │   └── __init__.py
│   ├── settings.ini
│   ├── settings.py
│   ├── web.ini
│   └── wsgi.py
├── test.py
├── tox.ini
└── webpack.config.jsdefault
```

где

- **frontend_src** - директория, содержащая все исходные файлы для фронтенда;
- **MANIFEST.in** - этот файл используется для создания установочного пакета;
- **{{app_name}}** - директория с файлами вашего приложения;
- **package.json** - этот файл содержит список зависимостей фронтенда и команд для сборки;
- **README.rst** - стандартный README файл для вашего приложения (этот файл включает базовые команды для запуска/остановки вашего приложения);
- **requirements-test.txt** - файл со списком зависимостей для тестирования вашего окружения;
- **requirements.txt** - файл со списком зависимостей вашего приложения;
- **setup.cfg** - файл, используемый для сборки установочного пакета;
- **setup.py** - файл, используемый для сборки установочного пакета;
- **test.py** - этот файл используется для создания тестов;
- **tox.ini** - этот файл используется для выполнения тестов;
- **webpack.config.js.default** - этот файл содержит минимальный скрипт для webpack (замените „.default“, если пишите что-то в „app.js“).

Вам нужно выполнять приведенные ниже команды из директории `/{{app_dir}}/{{app_name}}/`. Хорошей практикой будет использование tox (должен быть установлен перед использованием) для создания отладочной среды для вашего приложения. Для этих целей рекомендуется использовать `tox -e contrib` в директории проекта, что автоматически создаст новое окружение с необходимыми зависимостями.

3. **Применение миграций**

   Давайте проверим, работает ли новый проект vstutils. Перейдите во внешний каталог `/{{app_dir}}/{{app_name}}`, если вы еще этого не сделали, и выполните следующую команду:

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

   Эта команда создаст базу данных SQLite (по умолчанию) с SQL схемой по умолчанию. VSTUTILS поддерживает все базы данных которые поддерживает Django.[1]

4. **Создание суперпользователя**

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

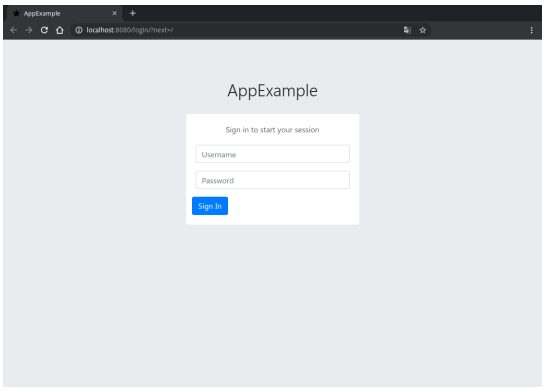
5. **Запуск приложения**

[1] https://docs.djangoproject.com/en/4.1/ref/databases/#databases

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

Веб-интерфейс вашего приложения будет запущен на порту 8080. Вы запустили сервер vstutils для продакшена на основе uWSGI[2].

Предупреждение: Сейчас хорошее время отметить: если вы хотите запустить веб-сервер с отладчиком, то вам следует запустить стандартный сервер разработки Django <https://docs.djangoproject.com/en/3.2/intro/tutorial01/#the-development-server>`_



Если вам нужно остановить сервер, используйте следующую команду:

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

Вы создали простейшее приложение, основанное на фреймворке VST Utils. Это приложение содержит только модель пользователя. Если вы хотите создать свои собственные модели, обратитесь к разделу ниже.

1.2 Добавление новых моделей в приложение

Если вы хотите добавить новые сущности в ваше приложение, вам необходимо выполнить следующие действия на серверной стороне:

1. Создайте модель;
2. Создайте сериализатор (опционально);
3. Создайте view (опционально);
4. Добавьте созданную модель или view в API;
5. Создайте миграции;
6. Примените миграции;
7. Перезапустите ваше приложение.

Давайте посмотрим, как это можно сделать на примере приложения AppExample которое содержит 2 пользовательские модели:

- Task (абстракция для некоторых задач/активностей, которые пользователь должен выполнить);
- Stage (абстракция для некоторых этапов, которые пользователь должен пройти для выполнения задачи. Эта модель вложена в модель Task).

[2] https://uwsgi-docs.readthedocs.io/

1.2.1 Создание моделей

Сначала вам необходимо создать файл `{{model_name}}.py` в директории `/{{app_dir}}/{{app_name}}/{{app_name}}/models`.

Давайте рассмотрим пример с моделью **BModel**:

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

Класс модели по умолчанию, который генерирует viewset, отдельные сериализаторы для list() и retrieve(), фильтры, эндпоинты API и вложенные view

Примеры:

```
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
            }
        }
```

В данном случае вы создаете модели, которые могут быть преобразованы в простое view, где:

- `POST`/`GET` по адресу `/api/version/task/` - создает новую задачу или получает список задач
- `PUT`/`PATCH`/`GET`/`DELETE` по адресу `/api/version/task/:id/` - обновляет, извлекает или удаляет экземпляр задачи
- `POST`/`GET` по адресу `/api/version/task/:id/stage/` - создает новую или получает список стадий в задаче
- `PUT`/`PATCH`/`GET`/`DELETE` по адресу `/api/version/task/:id/stage/:stage_id` - обновляет, извлекает или удаляет экземпляр стадии в задаче.

Для привязки view к API вставьте следующий код в файл *settings.py*:

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

Для основного доступа к сгенерированному view унаследуйтесь от свойства *Task.generated_view*.

Чтобы упростить перевод на фронтенде используйте атрибут `_translate_model` с model_name

Список мета-атрибутов для создания view:

- `_view_class` - список дополнительных классов view для наследования, класс или строка для импорта с базовым классом viewSet. Также поддерживаются константы:
  - `read_only` - для создания view только для просмотра;
  - `list_only` - для создания view только со списком;
  - `history` - для создания view только для просмотра и удаления записей

  CRUD-view применяется по умолчанию.
- `_serializer_class` - класс сериализатора API; используйте этот атрибут, чтобы указать родительский класс для автоматически создаваемых сериализаторов. По умолчанию используется *vstutils.api.serializers.VSTSerializer*. Может принимать строку для импорта, класс сериализатора или `django.utils.functional.SimpleLazyObject`.
- `_serializer_class_name` - имя модели для определений OpenAPI. Это будет имя модели в сгенерированном интерфейсе администратора. По умолчанию используется имя класса модели.
- `_list_fields` или `_detail_fields` - список полей, которые будут перечислены списке сущностей или детальном view соответственно. То же самое, что и мета-атрибут «fields» сериализаторов DRF.
- `_override_list_fields` или `_override_detail_fields` - отображение с именами и типами полей, которые будут переопределены в атрибутах сериализатора (рассматривайте это как объявление полей в сериализаторе DRF ModelSerializer).
- `_properties_groups` - словарь с ключом в виде имени группы и значением в виде списка полей (строк). Позволяет группировать поля в разделы на фронтенде.
- `_view_field_name` - имя поля, которое фронтенд показывает в качестве основного имени view.
- `_non_bulk_methods` - список методов, которые не должны использоваться через пакетные запросы.
- `_extra_serializer_classes` - отображение с дополнительными сериализаторами в viewset. Например, пользовательский сериализатор, который будет вычислять что-то в действии (имя отображения). Значение может быть строкой для импорта. Важное замечание: установка

атрибута модели в None позволяет использовать стандартный механизм создания сериализатора и получать поля из сериализатора списка или детальной записи (установите *__inject_from__* мета-атрибут сериализатора в *list* или *detail* соответственно). В некоторых случаях требуется передать модель в сериализатор. Для этого можно использовать константу *LAZY_MODEL* в качестве мета-атрибута. Каждый раз при использовании сериализатора будет установлена точная модель, в которой был объявлен этот сериализатор.

- `_filterset_fields` - список/словарь имен фильтров для фильтрации API. По умолчанию это список полей в view списке. При обработке списка полей проверяется наличие специальных имен полей и наследуются дополнительные родительские классы. Если список содержит `id`, класс будет наследоваться от *vstutils.api.filters.DefaultIDFilter*. Если список содержит `name`, класс будет наследоваться от *vstutils.api.filters.DefaultNameFilter*. Если присутствуют оба условия, наследование будет происходить от всех вышеперечисленных классов. Возможные значения включают список (*list*) полей для фильтрации или словарь (*dict*), где ключ - это имя поля, а значение - класс Filter. Словарь расширяет функциональность атрибута и позволяет переопределять класс поля фильтра (значение None отключает переопределение).
- `_search_fields` - кортеж или список полей, используемых для поисковых запросов. По умолчанию (или *None*) получаются все поля, по которым можно фильтровать в детальном view.
- `_copy_attrs` - список атрибутов экземпляра модели, указывающих, что объект можно скопировать с этими атрибутами.
- `_nested` - отображение ключ-значение с вложенными view (ключ - имя вложенного view, kwargs для декоратора *vstutils.api.decorators.nested_view*, но поддерживает атрибут `model` в качестве вложенного view). `model` может быть строкой для импорта.
- `_extra_view_attributes` - отображение ключ-значение с дополнительными атрибутами view, но имеет меньший приоритет перед сгенерированными атрибутами.

Также вы также можете добавлять пользовательские атрибуты для переопределения или расширения списка обрабатываемых классов по умолчанию. Поддерживаемые атрибуты view: `filter_backends`, `permission_classes`, `authentication_classes`, `throttle_classes`, `renderer_classes` и `parser_classes`. Список мета-атрибутов для настроек view выглядит следующим образом:

- `_pre_{attribute}` - список классов, включаемых перед значениями по умолчанию.
- `_{attribute}` - список классов, включаемых после значений по умолчанию.
- `_override_{attribute}` - флаг-булево значение, указывающее, переопределяет ли атрибут view по умолчанию (в противном случае добавляется). По умолчанию False.

Примечание: Возможно, вам понадобится создать action[3] в сгенерированном view. Используйте декоратор *vstutils.models.decorators.register_view_action* с аргументом `detail`, чтобы определить применимость к списку или детальной записи. В этом случае декорированный метод будет принимать объект view в качестве атрибута `self`.

Примечание: В некоторых случаях при наследовании моделей может потребоваться наследовать класс Meta от базовой модели. Если Meta явно объявлена в базовом классе, вы можете получить ее через атрибут *OriginalMeta* и использовать для наследования.

Примечание: Docstring модели будет использоваться для описания view. Можно написать как общее описание для всех действий, так и описание для каждого действия, используя следующий синтаксис:

```
General description for all actions.

action_name:
    Description for this action.

another_action:
    Description for another action.
```

Более подробную информацию о моделях вы можете найти в документации Django Models[4].

Если вам не нужно создавать пользовательские *сериализаторы* или *view sets*, вы можете перейти к этому *этапу*.

1.2.2 Создание сериализаторов

Примечание - Если вам не нужен пользовательский сериализатор, вы можете пропустить этот раздел.

В первую очередь вам необходимо создать файл `serializers.py` в директории `/{{app_dir}}/{{app_name}}/{{app_name}}/`.

Затем вам нужно добавить некоторый код, подобный следующему, в файл `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)
```

Более подробную информацию о сериализаторах вы можете найти в документации Django REST Framework по сериализаторам[5].

1.2.3 Создание views

Примечание - Если вам не нужен пользовательский view set, вы можете пропустить этот раздел.

В первую очередь вам необходимо создать файл `views.py` в директории `/{{app_dir}}/{{app_name}}/{{app_name}}/`.

Затем вам нужно добавить некоторый код, подобный следующему, в файл `views.py`:

[3] https://www.django-rest-framework.org/api-guide/viewsets/#marking-extra-actions-for-routing
[4] https://docs.djangoproject.com/en/3.2/topics/db/models/
[5] https://www.django-rest-framework.org/api-guide/serializers/#modelserializer

```
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.
    '''
```

Больше информации о view и viewset вы можете найти в документации Django REST Framework для view[6].

1.2.4 Добавление моделей в API

Для добавления моделей в APi вам нужно написать код, подобный этому в в конце файла `settings.py`:

```
'''
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.
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'
)
```

(continues on next page)

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

(продолжение с предыдущей страницы)

```
# 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 Создание миграций

Для создания миграций откройте директорию `/{{app_dir}}/{{app_name}}/` и выполните следующую команду:

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

Более подробную информацию о миграциях вы можете найти в документации Django Migrations[7].

1.2.6 Применение миграций

Для применения миграций вам необходимо открыть директорию `/{{app_dir}}/{{app_name}}/` и выполнить следующую команду:

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

1.2.7 Перезапуск приложения

Для перезапуска вашего приложения вам сначала нужно остановить его (если оно было запущено ранее):

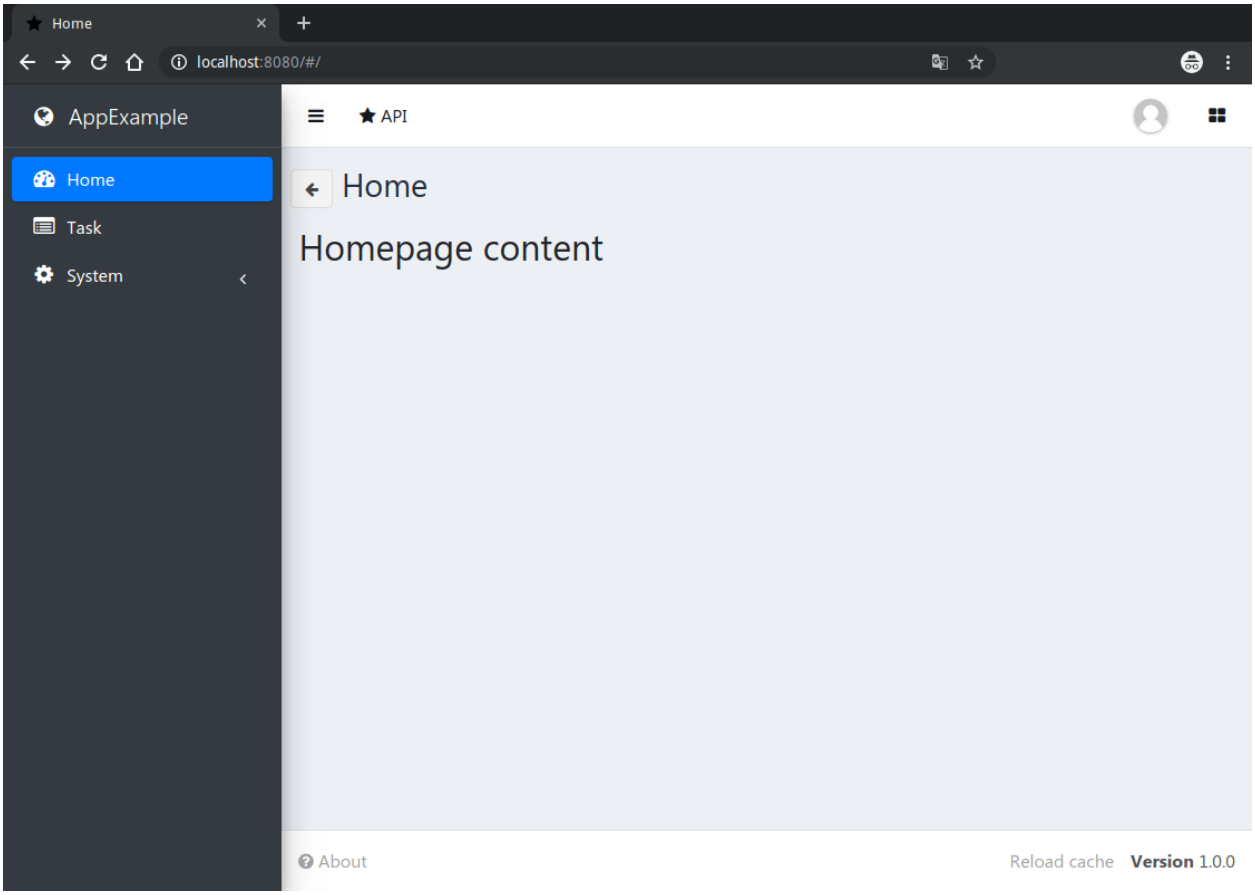
```
python -m {{app_name}} web stop=/tmp/{{app_name}}_web.pid
```

Затем запустите его снова:

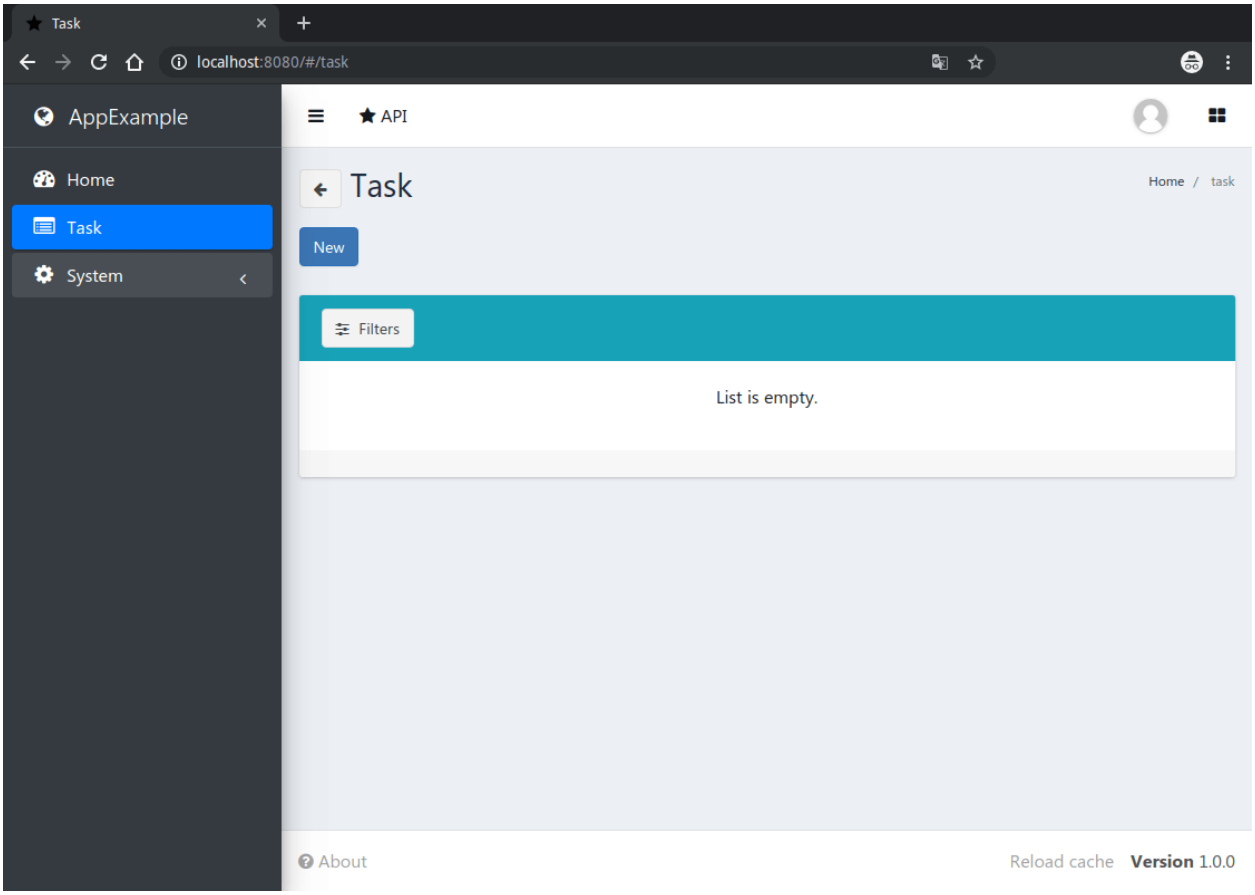
```
python -m {{app_name}} web
```

После перезагрузки кэша вы увидите следующую страницу:

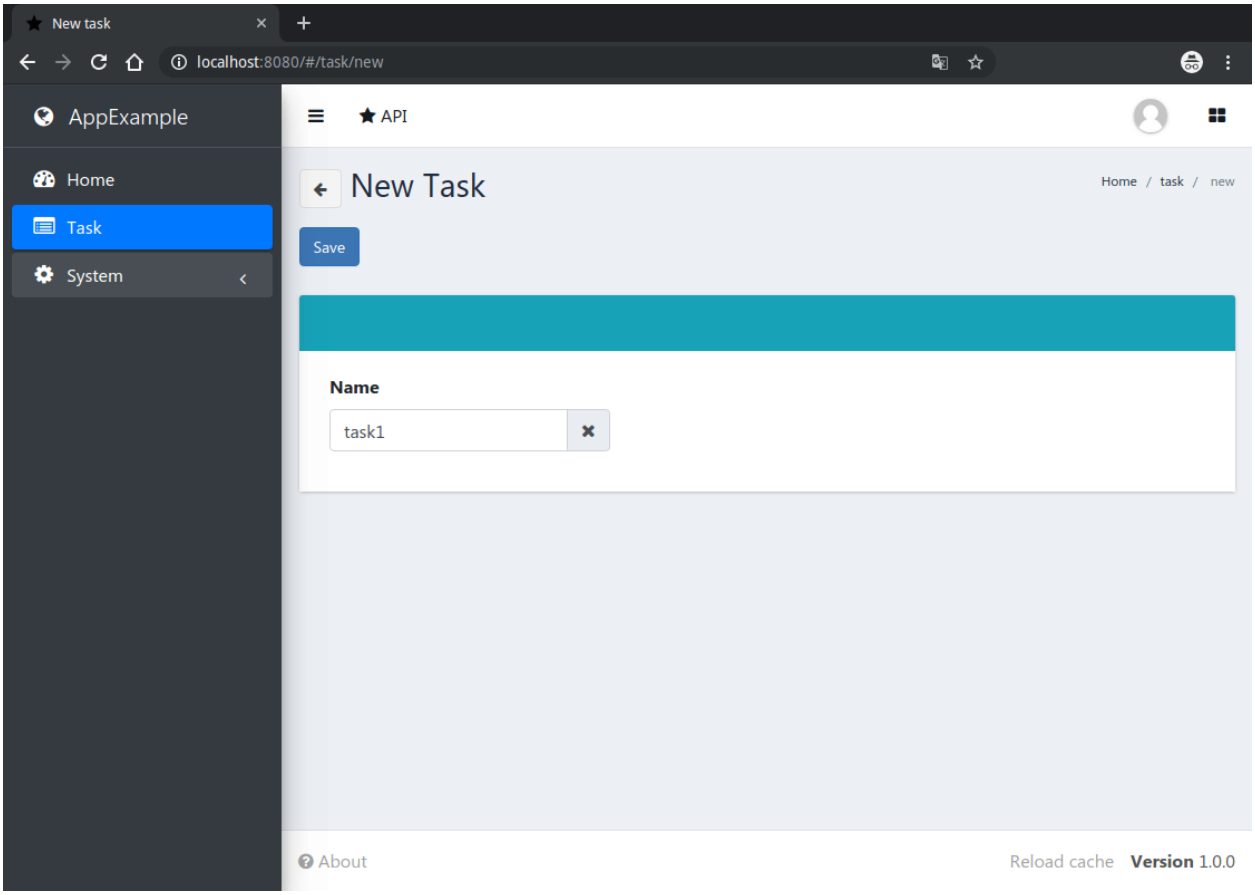
[7] https://docs.djangoproject.com/en/3.2/topics/migrations/



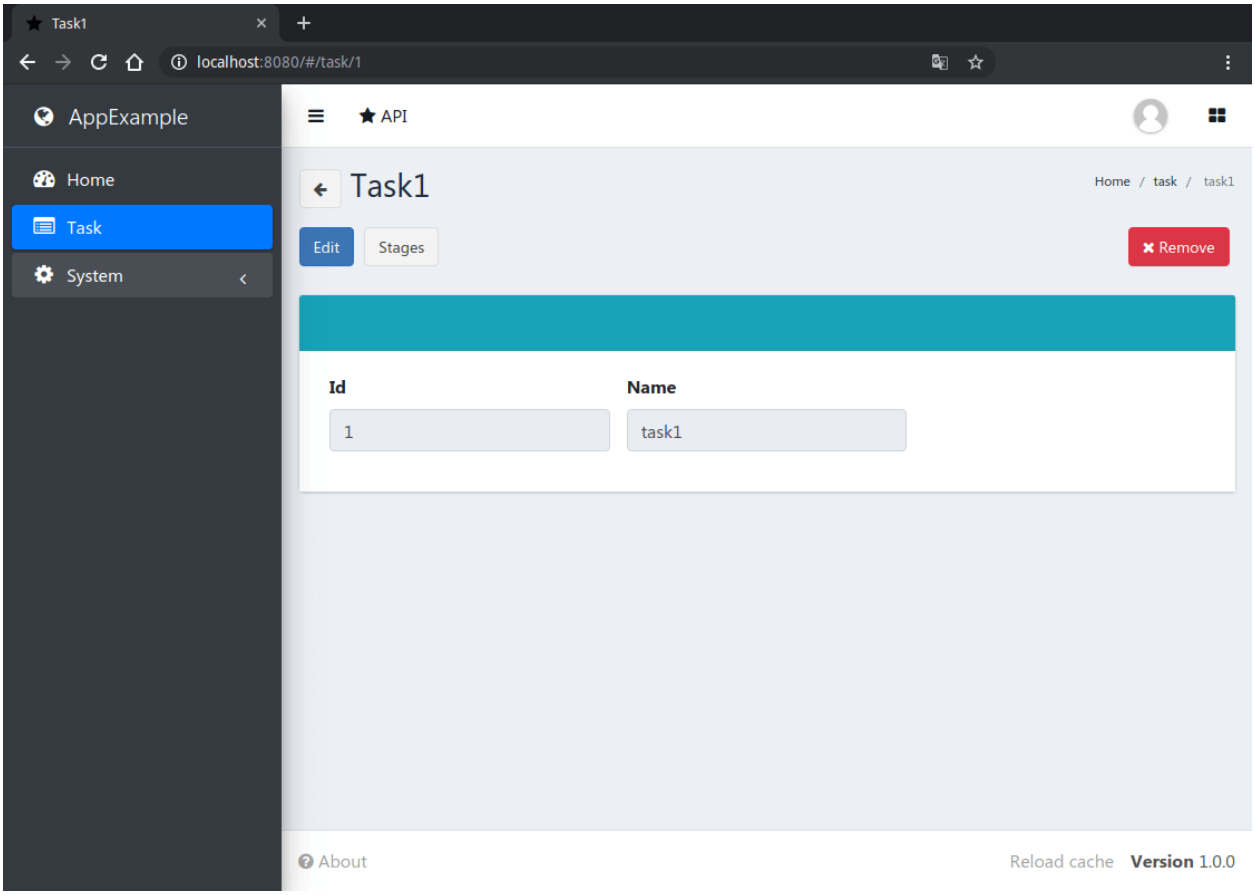
Как вы можете видеть, ссылка на новое Task view добавлена в боковое меню. Давайте нажмем на нее.



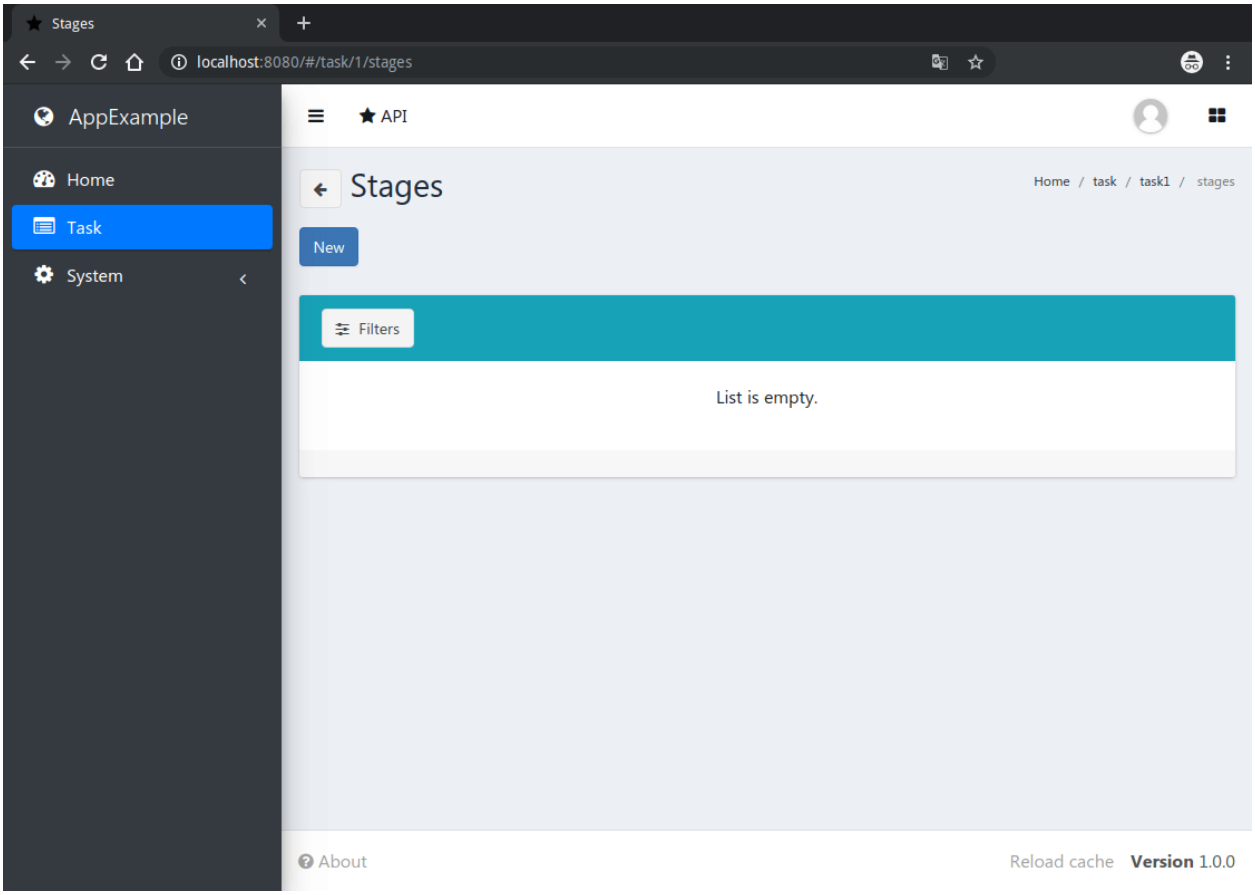
В вашем приложении нет экземпляра задачи. Добавьте его, используя кнопку „new“.



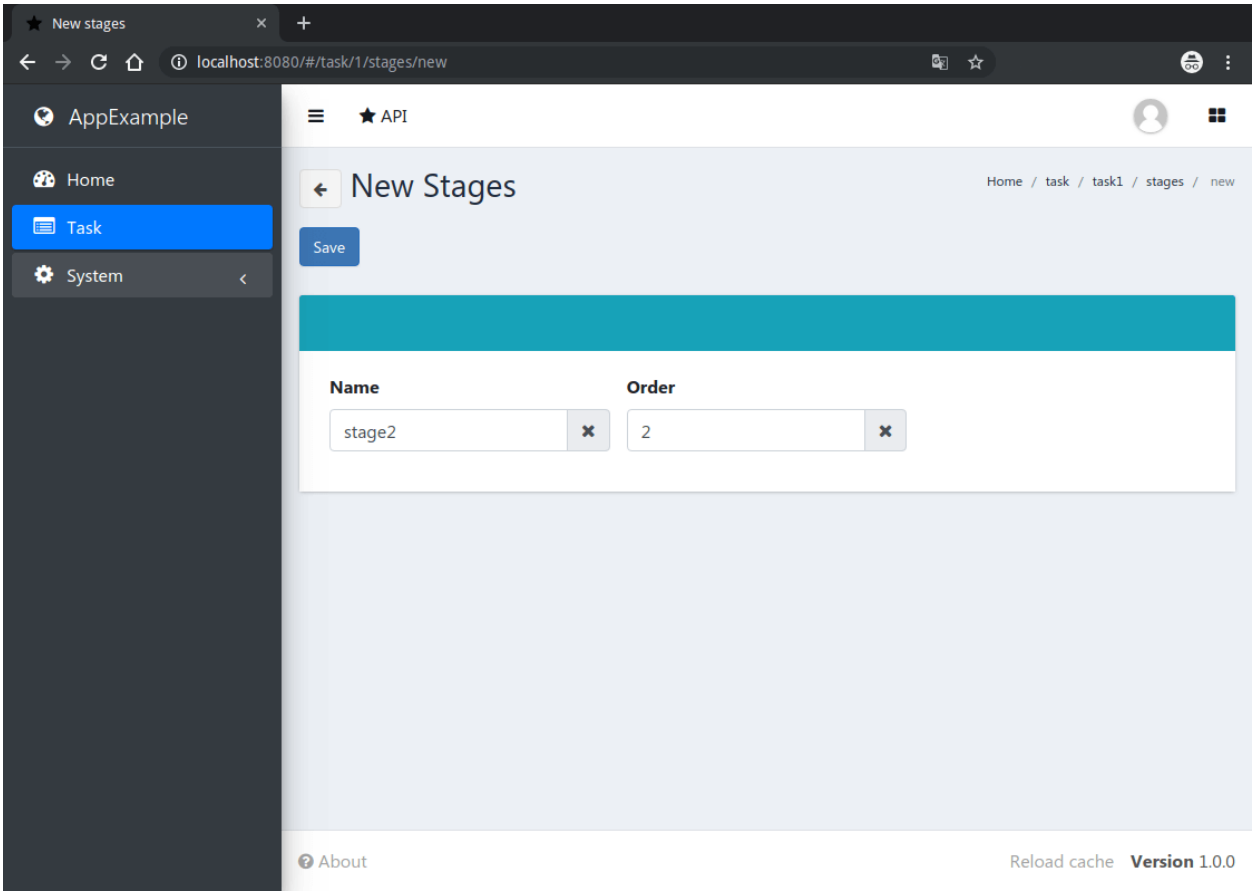
После создания новой задачи вы увидите следующую страницу:

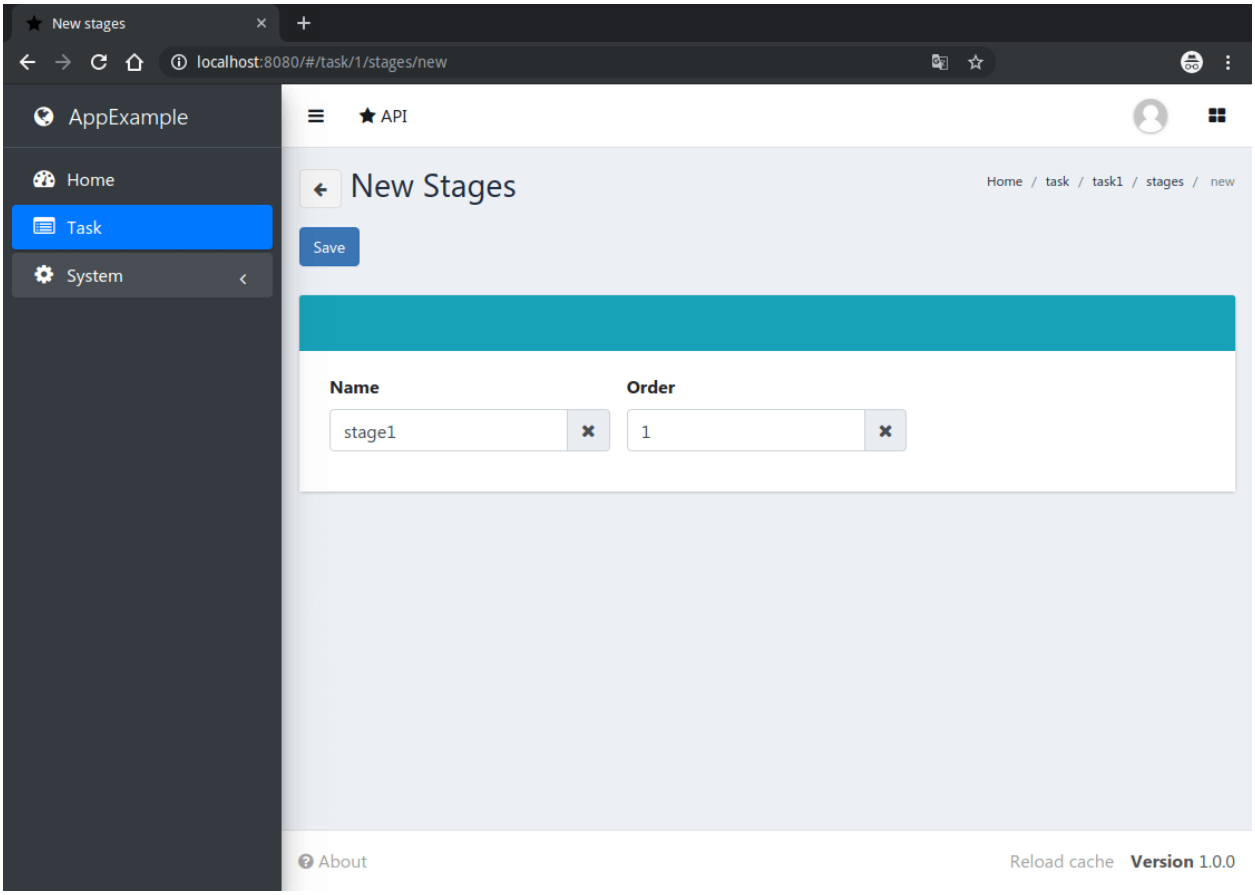


Как видите, есть кнопка „stages“, которая открывает страницу со списком этапов этой задачи. Давайте на нее нажмем.

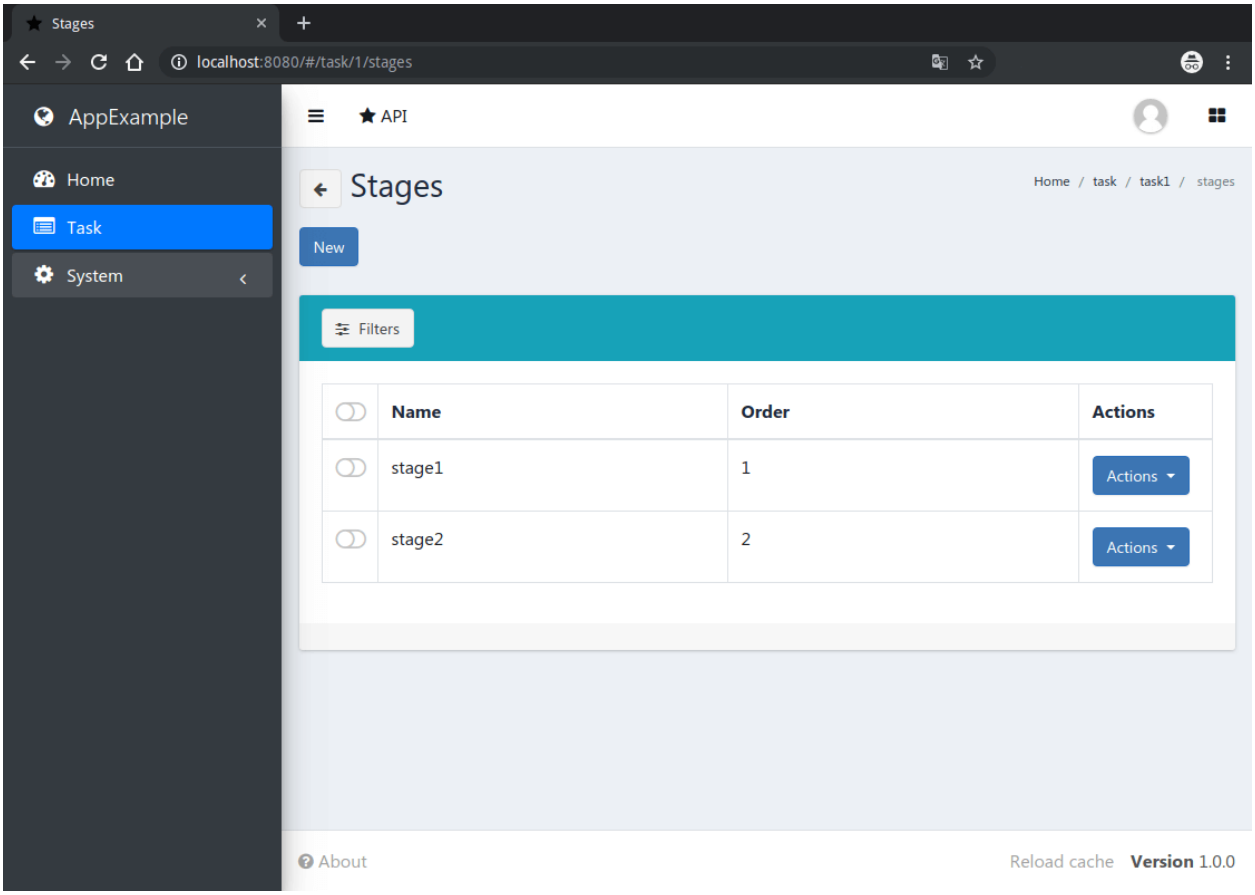


В вашем приложении нет экземпляров этапов. Давайте создадим 2 новых этапа.


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



После создания этапов страница со списком этапов будет выглядеть так:



Сортировка по полю *order* работает, как мы указали в нашем файле `models.py` для модели Stage.

Дополнительную информацию о Django и Django REST Framework вы можете найти в документации Django[8] и документации Django REST Framework[9].

[8] https://docs.djangoproject.com/en/3.2/
[9] https://www.django-rest-framework.org/

2 Руководство по настройке

2.1 Введение

Хоть и стандартная конфигурация подходит в большинстве случаев, приложение на vstutils является высоко настраиваемой системой. Для расширенных настроек (масштабируемость, выделенная база данных, настраиваемый кэш, логгирование или директории) вы можете глубоко настраивать приложение, основанное на vstutils, изменяя файл настроек `/etc/{{app_name or app_lib_name}}/settings.ini`.

Самое важное, о чем нужно помнить при планировании архитектуры вашего приложения, это то, что приложения, основанные на vstutils, имеют сервисно-ориентированную структуру. Чтобы построить распределенную масштабируемую систему, вам нужно только подключиться к *общей базе данных*, *общему кэшу*, *блокировкам* и общей *службе rpc* (MQ, такому как RabbitMQ, Redis и т. д.). В некоторых случаях может потребоваться общее файловое хранилище, но vstutils не требует его.

Рассмотрим основные разделы конфигурации и их параметры:

2.2 Основные настройки

Раздел `[main]`.

Этот раздел предназначен для настроек, относящихся ко всему приложению на основе vstutils (как рабочему процессу, так и веб-интерфейсу). Здесь вы можете указать уровень подробности вывода информации о работе приложения на основе vstutils, что может быть полезно при устранении неполадок (уровень логгирования и т. д.). Также здесь находятся настройки для изменения часового пояса приложения в целом и разрешенных доменов.

Чтобы использовать протокол LDAP, создайте следующие настройки в разделе `[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 - это необязательный аргумент, который направлен на то, чтобы сделать авторизацию проще (без ввода доменного имени).

ldap-auth_format это необязательный аргумент, который направлен на то, чтобы кастомизировать LDAP авторизацию. Значение по умолчанию: cn=<username>,<domain>

В примере выше логика авторизации будет следующая:

1. Система проверяет комбинацию login:password в базе данных
2. Система проверяет комбинацию login:password в LDAP:
    - Если домен был указан, то он будет установлен во время авторизации (например, если пользователь ввел login без *user@domain.name*` или без `DOMAIN\user`);
    - Если авторизация была успешной и пользователь с предоставленными данными существует в базе данных, сервер создает сессию этого пользователя.

- **debug** - Включить режим отладки. По умолчанию: false.
- **allowed_hosts** - Разделенный запятой список доменов, которым разрешено обслуживание. По умолчанию: `*`.
- **first_day_of_week** - Целочисленное значение первого дня недели. по умолчанию: `0`.
- **ldap-server** - Подключение к серверу LDAP.
- **ldap-default-domain** - Домен по умолчанию для аутентификации
- **ldap-auth_format** - Формат запроса поиска по умолчанию для аутентификации. По умолчанию: `cn=<username>,<domain>`.
- **timezone** - часовой пояс для веб-приложения. По умолчанию: UTC.
- **log_level** - Уровень логгирования. По умолчанию WARNING.
- **enable_django_logs** - Включить или выключить вывод логов Django. Полезно для отладки. По умолчанию: false.
- **enable_admin_panel** - Включить или выключить панели администратора Django. По умолчанию: false.
- **enable_registration** - Включить или выключить самостоятельную регистрацию пользователей. По умолчанию: false.
- **enable_user_self_remove** - Включить или выключить самоудаление пользователей. По умолчанию: false.
- **auth-plugins** - Список аутентифицированных бэкендов Django, разделенных запятыми. Попытка авторизации повторяется до первой успешной в соответствии с порядком, указанным в списке.
- **auth-cache-user** - Включить или выключить кэширование экземпляра пользователя. Это увеличивает производительность сеанса при каждом запросе, но сохраняет экземпляр модели в небезопасном хранилище (кэше Django по умолчанию). Экземпляр сериализуется в строку с использованием `стандартного модуля python pickle`[10], а затем шифруется с помощью шифра Виженера[11]. Дополнительную информацию можно найти в документации *`vstutils.utils.SecurePickling`*. По умолчанию: false.

2.3 Настройки базы данных

Раздел `[databases]`.

Основной раздел, предназначенный для управления несколькими базами данных, которые подключены к проекту.

Эти настройки актуальны для всех баз данных за исключением пространства таблиц.

- **default_tablespace** - Табличное пространство по умолчанию, используемое для индексов полей, в которых оно не указано, если бэкенд это поддерживает. Дополнительную информацию можно найти в разделе Объявление табличных пространств для индексов[12].

[10] https://docs.python.org/3.6/library/pickle.html#module-pickle
[11] https://en.wikipedia.org/wiki/Vigenèrecipher
[12] https://docs.djangoproject.com/en/4.2/topics/db/tablespaces/#declaring-tablespaces-for-indexes

- **default_index_tablespace** - Табличное пространство по умолчанию, используемое для индексов полей, в которых оно не указано, если бэкенд это поддерживает Дополнительную информацию можно найти в разделе Объявление табличных пространств для индексов[13].»
- **databases_without_cte_support** - Разделенный запятыми список разделов баз данных которые не поддерживают CTEs (Common Table Expressions).

Предупреждение: Хоть MariaDB и поддерживает CTEs (Common Table Expressions), но база данных, подключенная к MariaDB все равно должна быть добавлена в список `databases_without_cte_support`. Проблема заключается в том, что реализация рекурсивных запросов MariaDB не позволяет использовать их в стандартной форме. MySQL (начиная с версии 8.0) работает ожидаемым образом.

Также, все подразделы этого раздела представляют собой доступные подключения к СУБД. Таким образом, раздел `databases.default` будет использоваться Django в качестве подключения по умолчанию.

Здесь вы можете изменять настройки, связанные с базой данных, которую будет использовать приложение, основанное на vstutils. Приложение, основанное на vstutils, поддерживает все базы данных, поддерживаемые `Django`. Список поддерживаемых баз данных изначально включает SQLite (выбор по умолчанию), MySQL, Oracle или PostgreSQL. Подробную информацию о конфигурации можно найти в документации Django по базам данных[14]. Для запуска приложения, основанного на vstutils, на нескольких узлах (кластере) используйте клиент-серверную базу данных (SQLite не подходит), используемую всеми узлами.

Вы также можете задать базовый шаблон для подключения к базе данныхв разделе `database`.

Раздел `[database]`.

Этот раздел предназначен для определения базового шаблона для подключения к различным базам данных. Это может быть полезно для сокращения списка настроек в подразделах `databases.*` путем установки одного и того же подключения для различного набора баз данных в проекте. Дополнительные сведения можно найти в документации Django о множественных базах данных[15]

Здесь приведен список настроек, необходимых для базы данных MySQL/MariaDB

Во-первых, если вы используете MySQL/MariaDB и установили часовой пояс, отличный от «UTC», вам следует выполнить следующую команду:

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

Во-вторых, чтобы использовать MySQL/MariaDB установите следующие настройки в файле `settings.ini`:

```
[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'
```

Наконец, добавьте некоторые настройки в конфигурацию MySQL/MariaDB

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

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

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

2.4 Настройки кэша

Раздел `[cache]`.

В этом разделе находятся настройки, связанные с кэшем, используемые приложением, основанным на vstutils. vstutils поддерживает все бэкэнды кэша, которые поддерживает Django. Файловая система, в памяти, memcached поддерживаются изначально, а многие другие поддерживаются с помощью дополнительных плагинов. Подробную информацию о настройках кэша можно найти в документации Django о поддерживаемых кэшах[16]. При кластеризации мы рекомендуем делить кэш между узлами для повышения производительности с использованием реализаций клиент-серверного кэша. Мы рекомендуем использовать Redis в производственных окружениях.

2.5 Настройки блокировок

Раздел `[locks]`.

Блокировки - это система, которую приложение, основанное на vstutils, использует для предотвращения повреждения от параллельных действий, выполняемых одновременно над одной сущностью. Она основана на кэше Django, поэтому есть еще одна группа настроек, аналогичных настройкам *cache*. Вы можете спросить, почему для них существует еще один раздел. Потому что бэкэнд кэша, используемый для блокировки, должен обеспечивать некоторые гарантии, которые не требуются для обычного кэша: он ДОЛЖЕН быть общим для всех потоков и узлов приложения, основанного на vstutils. Например, бэкенд in-memory не подходит. В случае кластеризации настоятельно рекомендуется использовать Redis или Memcached в качестве бэкэнда для этой цели. Бэкэнд кэша и блокировок могут быть одним и тем же, но не забывайте о требованиях, о которых мы говорили выше.

2.6 Настройки кэша сессий

Раздел `[session]`.

Приложение, основанное на vstutils, хранит сеансы в *базе данных*, но для повышения производительности мы используем кэшевый бэкэнд для сеансов. Он также основан на кэше Django, поэтому здесь есть еще одна группа настроек, аналогичных настройкам *cache*. По умолчанию настройки получаются из *cache*.

2.7 Настройки RPC

Раздел `[rpc]`.

Приложение, основанное на vstutils, использует Celery для выполнения длительных асинхронных задач. Celery основан на концепции очереди сообщений, поэтому между веб-сервисом и рабочими процессами, работающими под управлением Celery, должен быть некий брокер сообщений (например, RabbitMQ). Эти настройки относятся к этому брокеру и самому Celery. В них указывается бэкэнд брокера, количество рабочих процессов на узел и некоторые настройки, используемые для устранения проблем взаимодействия сервера, брокера и рабочих процессов.

Для этого раздела требуется vstutils с дополнительной зависимостью rpc.

- **connection** - Соединение с брокером celery[17]. По умолчанию: `filesystem:///var/tmp`.
- **concurrency** - Количество потоков рабочего процесса Celery. По умолчанию: 4.

[16] https://docs.djangoproject.com/en/4.2/ref/settings/#caches
[17] http://docs.celeryproject.org/en/latest/userguide/configuration.html#conf-broker-settings

- **heartbeat** - Интервал между отправкой пакетов-сигналов, которые говорят, что соединение все еще активно. По умолчанию: 10.
- **enable_worker** - Включить или отключить рабочий процесс с веб-сервером. По умолчанию: true.

Также поддерживаются следующие переменные из настроек Django[18] (с соответствующими типами):

- **prefetch_multiplier** - CELERYD_PREFETCH_MULTIPLIER[19]
- **max_tasks_per_child** - CELERYD_MAX_TASKS_PER_CHILD[20]
- **results_expiry_days** - CELERY_RESULT_EXPIRES[21]
- **default_delivery_mode** - CELERY_DEFAULT_DELIVERY_MODE[22]
- **task_send_sent_event** - CELERY_DEFAULT_DELIVERY_MODE[23]
- **worker_send_task_events** - CELERY_DEFAULT_DELIVERY_MODE[24]

2.8 Настройки рабочего процесса (worker`a)

Раздел `[worker]`.

Предупреждение: Эти настройки необходимы только для приложений с включенной поддержкой RPC.

Настройки рабочего процесса celery:

- **loglevel** - Уровень логгирования рабочего процесса. По умолчанию: из раздела *main* `log_level`.
- **pidfile** - Файл pid для рабочего процесса Celery. по умолчанию: `/run/{app_name}_worker.pid`»
- **autoscale** - Параметры для автомасштабирования. Два числа, разделенных запятой: максимальное,минимальное.
- **beat** - Включить или отключить планировщик celery beat. По умолчанию: `true`.

Другие настройки можно увидеть с помощью команды `celery worker --help`

2.9 SMTP-настройки

Раздел `[mail]`.

Django поставляется с несколькими вариантами отправки электронной почты. За исключением бэкэнда SMTP (который является значением по умолчанию при установке `host`), эти бэкэнды полезны только для тестирования и разработки.

Приложения, основанные на vstutils, используют только бэкэнды `smtp` и `console`.

- **host** - IP-адрес или доменное имя smtp-сервера. Если не указано, vstutils использует бэкэнд `console`. По умолчанию: `None`.

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

- **port** - Порт для подключения к smtp-серверу. По умолчанию: `25`.
- **user** - Имя пользователя для подключения к SMTP-серверу. По умолчанию: `""`.
- **password** - Пароль для аутентификации на smtp-сервере. По умолчанию: `""`.
- **tls** - Включить или отключить TLS для подключения к smtp-серверу. По умолчанию: `False`
- **send_confirmation** - Включить или отключить отправку сообщения с подтверждением после регистрации. По умолчанию: `False`.
- **authenticate_after_registration** - Включить или отключить автоматический вход пользователя после подтверждения регистрации. По умолчанию: `False`.

2.10 Web-настройки

Раздел `[web]`.

Эти настройки относятся к веб-серверу. Среди них: session_timeout, static_files_url и лимит пагинации.

- **allow_cors** - включить cross-origin resource sharing. По умолчанию: `False`.
- **cors_allowed_origins**, **cors_allowed_origins_regexes**, **cors_expose_headers**, **cors_allow_methods**, **cors_allow_headers**, **cors_preflight_max_age** - Настройки[25] из библиотеки `django-cors-headers` со значениями по умолчанию.
- **enable_gravatar** - Включить/отключить использование сервиса Gravatar для пользователей. По умолчанию: `True`.
- **rest_swagger_description** - Строка справки в схеме Swagger. Полезно для разработки интеграций.
- **openapi_cache_timeout** - Время кэширования данных схемы. По умолчанию: `120`.
- Количество запросов к конечной точке `/api/health/`. По умолчанию: `60`.
- **bulk_threads** - Количество потоков для PATCH `/api/endpoint/`. По умолчанию: `3`.
- **session_timeout** - Время жизни сессии. По умолчанию: `2w` (две недели).
- **etag_default_timeout** - Время кэширования заголовков Etag для управления кэшированием моделей. По умолчанию: `1d` (один день).
- **rest_page_limit** and **page_limit** - Максимальное количество объектов в списке API. По умолчанию: `1000`.
- **session_cookie_domain** - Домен, используемый для сессионных cookie. Подробнее[26]. По умолчанию: `None`.
- **csrf_trusted_origins** - Список хостов, которым доверяются небезопасные запросы. Подробнее[27]. По умолчанию: значение из **session_cookie_domain**.
- **case_sensitive_api_filter** - Включить/отключить чувствительность к регистру при фильтрации по имени. По умолчанию: `True`.
- **secure_proxy_ssl_header_name** - Имя заголовка, которое активирует использование URL-адресов SSL в ответах. Подробнее[28]. По умолчанию: `HTTP_X_FORWARDED_PROTOCOL`.
- **secure_proxy_ssl_header_value** - Значение заголовка, которое активирует использование URL-адресов SSL в ответах. Подробнее[29]. По умолчанию: `https`.

[25] https://github.com/adamchainz/django-cors-headers#configuration
[26] https://docs.djangoproject.com/en/4.2/ref/settings/#std:setting-SESSION_COOKIE_DOMAIN
[27] https://docs.djangoproject.com/en/4.2/ref/settings/#csrf-trusted-origins
[28] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-proxy-ssl-header
[29] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-proxy-ssl-header

Также поддерживаются следующие переменные из настроек Django (с соответствующими типами):

- **secure_browser_xss_filter** - SECURE_BROWSER_XSS_FILTER[30]
- **secure_content_type_nosniff** - SECURE_CONTENT_TYPE_NOSNIFF[31]
- **secure_hsts_include_subdomains** - SECURE_HSTS_INCLUDE_SUBDOMAINS[32]
- **secure_hsts_preload** - SECURE_HSTS_PRELOAD[33]
- **secure_hsts_seconds** - SECURE_HSTS_SECONDS[34]
- **password_reset_timeout_days** - PASSWORD_RESET_TIMEOUT_DAYS[35]
- **request_max_size** - DATA_UPLOAD_MAX_MEMORY_SIZE[36]
- **x_frame_options** - X_FRAME_OPTIONS[37]
- **use_x_forwarded_host** - USE_X_FORWARDED_HOST[38]
- **use_x_forwarded_port** - USE_X_FORWARDED_PORT[39]

Следующие настройки влияют на эндпоинт метрик Prometheus (который может использоваться для мониторинга приложения):

- **metrics_throttle_rate** - Количество запросов к эндпоинту `/api/metrics/`. По умолчанию: `120`.
- **enable_metrics** - Включить/отключить эндпоинт `/api/metrics/` для приложения. По умолчанию: `true`.
- **metrics_backend** - Путь к классу Python с бэкендом сборщика метрик. По умолчанию: `vstutils.api.metrics.DefaultBackend`. Стандартный бэкенд собирает метрики из рабочих процессов uwsgi и информацию о версии Python.

Раздел `[uvicorn]`.

Вы можете настроить необходимые параметры для запуска сервера uvicorn. `vstutils` поддерживает практически все опции из командной строки, за исключением тех, которые настраивают приложение и соединение.

Вы можете посмотреть все доступные настройки uvicorn, введя команду `uvicorn --help`

2.11 Настройки клиента Centrifugo

Раздел `[centrifugo]`.

Для установки приложения с клиентом Centrifugo должен быть задан раздел `[centrifugo]`. Centrifugo используется приложением для автоматического обновления данных на странице. Когда пользователь изменяет какие-либо данные, другие клиенты получают уведомление на канале `subscriptions_update` с меткой модели и первичным ключом. Без этой службы все клиенты GUI получают данные страницы каждые 5 секунд (по умолчанию).

- **address** - Адрес сервера Centrifugo.

[30] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-browser-xss-filter
[31] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-content-type-nosniff
[32] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-hsts-include-subdomains
[33] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-hsts-preload
[34] https://docs.djangoproject.com/en/4.2/ref/settings/#secure-hsts-seconds
[35] https://docs.djangoproject.com/en/4.2/ref/settings/#std:setting-PASSWORD_RESET_TIMEOUT
[36] https://docs.djangoproject.com/en/4.2/ref/settings/#std:setting-DATA_UPLOAD_MAX_MEMORY_SIZE
[37] https://docs.djangoproject.com/en/4.2/ref/settings/#x-frame-options
[38] https://docs.djangoproject.com/en/4.2/ref/settings/#use-x-forwarded-host
[39] https://docs.djangoproject.com/en/4.2/ref/settings/#use-x-forwarded-port

- **api_key** - Ключ API для клиентов.
- **token_hmac_secret_key** - Ключ API для генерации JWT-токена.
- **timeout** - Таймаут подключения.
- **verify** - Проверка подключения.
- **subscriptions_prefix** - Префикс, используемый для генерации каналов обновления, по умолчанию «{VST_PROJECT}.update».

Примечание: Эти настройки также добавляют параметры в схему OpenAPI и меняют работу системы автообновления в GUI. `token_hmac_secret_key` используется для генерации JWT-токена (на основе времени истечения сессии). Токен будет использоваться для клиента Centrifugo-JS.

2.12 Настройки хранилища

Раздел `[storages]`.

Приложения, основанные на `vstutils`, поддерживают хранение файлов в файловой системе из коробки. Чтобы настроить пользовательский медиа-каталог и относительный URL, установите значения `media_root` и `media_url` в разделе [storages.filesystem]. По умолчанию они будут равны `{/path/to/project/module}/media` и `/media/`.

Приложения, основанные на `vstutils`, также поддерживают хранение файлов во внешних службах с помощью Apache Libcloud[40] и Boto3[41].

Настройки Apache Libcloud группируются по разделам с именами `[storages.libcloud.provider]`, где `provider` - это имя хранилища. Каждый раздел имеет четыре ключа: `type`, `user`, `key` и `bucket`. Подробнее о настройках можно прочитать в документации django-storages libcloud[42].

Эта настройка необходима для настройки соединений с провайдерами облачного хранилища. Каждая запись соответствует отдельному 'bucket' хранилища. Вы можете иметь несколько buckets для одного провайдера услуг (например, несколько buckets S3), и вы можете определить buckets в нескольких провайдерах.

Для `Boto3` все настройки группируются в разделе с именем `[storages.boto3]`. Раздел должен содержать следующие ключи: `access_key_id`, `secret_access_key`, `storage_bucket_name`. Подробнее о настройках можно прочитать в документации django-storages amazon-S3[43].

При выборе движка хранилища используется следующий приоритет, если их было предоставлено несколько:

1. Хранилище Libcloud, когда конфигурация содержит этот раздел.
2. Хранилище Boto3, когда у вас есть раздел и имеются все необходимые ключи.
3. В противном случае хранилище FileSystem.

После того, как вы определили ваших провайдеров Libcloud, у вас есть возможность установить одного, как провайдера по умолчанию для хранилища Libcloud. Вы можете сделать это, настроив раздел `[storages.libcloud.default]` или же vstutils установит первое хранилище, как хранилище по умолчанию.

Если вы настроили провайдера Libcloud по умолчанию, vstutils будет использовать его в качестве глобального хранилища файлов. Чтобы переопределить это, установите `default=django.core.files.storage.FileSystemStorage` в разделе `[storages]`. Когда `[storages.libcloud.default]` пу-

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

сто, по умолчанию используется `django.core.files.storage.FileSystemStorage`. Чтобы переопределить это, установите `default=storages.backends.apache_libcloud.LibCloudStorage` в разделе `[storages]` и используйте провайдера Libcloud, как по умолчанию.

Вот пример подключения boto3 к кластеру minio с публичными правами на чтение, внешним доменом прокси и поддержкой внутреннего подключения:

```
[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/
# 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 Настройки Throttle

Раздел `[throttle]`.

Путем добавления этой секции в вашу конфигурацию вы можете настроить глобальные и индивидуальные throttle rates для каждого View. Глобальные throttle rates указываются в секции [throttle]. Чтобы указать индивидуальные throttle rates для конкретного View, вам нужно добавить дочернюю секцию.

Например, если вы хотите применить ограничение количества запросов для `api/v1/author`:

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

- **rate** - Ограничение количества запросов в формате number_of_requests/time_period. Expected time_periods: second/minute/hour/day.
- **actions** - Разделенный запятой список действий DRF. Ограничение количества запросов будет применяться только к указанным здесь действиям. По умолчанию: update, partial_update.

Подробнее об ограничении количества запросов в документации DRF Throttle[44].

[44] https://www.django-rest-framework.org/api-guide/throttling/

2.14 Настройки для продакшн-сервера

Раздел `[uwsgi]`.

Настройки, связанные с веб-сервером, используемым в приложении на основе vstutils в продакшн-среде (по умолчанию для пакетов deb и rpm). Большинство из них относятся к системным путям (логгирование, PID-файл и т. д.). Дополнительные настройки смотрите в документации uWSGI.[45].

Однако имейте в виду, что uWSGI устарел и может быть удален в будущих версиях. Используйте настройки uvicorn для управления сервером вашего приложения.

2.15 Параметры конфигурации

В этом разделе содержится дополнительная информация для настройки дополнительных элементов.

1. Если вам необходимо настроить `HTTPS` для ваших веб-настроек, вы можете сделать это с помощью HAProxy, Nginx, Traefik или настроить в файле `settings.ini`.

```
[uwsgi]
addrport = 0.0.0.0:8443

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

1. Мы настоятельно не рекомендуем запускать веб-сервер от имени root. Используйте HTTP-прокси, чтобы работать на привилегированных портах.
2. Вы можете использовать *{ENV[HOME:-value]}* (где *HOME* - переменная окружения, *value* - значение по умолчанию) в значениях конфигурации.
3. Вы можете использовать переменные окружения для настройки важных параметров. Однако переменные конфигурации имеют более высокий приоритет, чем переменные окружения. Доступные настройки: `DEBUG`, `DJANGO_LOG_LEVEL`, `TIMEZONE` и некоторые настройки с префиксом `[ENV_NAME]`.

   Для проекта без специальных настроек и проектов с именами, начинающимися с `project`, эти переменные будут иметь префикс `PROJECT_`. Вот список этих переменных: `{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` (директория с загрузками), `{ENV_NAME}_GLOBAL_THROTTLE_RATE`, и `{ENV_NAME}_GLOBAL_THROTTLE_ACTIONS`.

   Также существуют переменные, специфичные для URI, для подключения к различным сервисам, таким как базы данных и кэши. Вот некоторые из них `DATABASE_URL`, `CACHE_URL`, `LOCKS_CACHE_URL`, `SESSIONS_CACHE_URL` и `ETAG_CACHE_URL`. Как видно из названий, они тесно связаны с ключами и именами соответствующих секций конфигурации.
4. Мы рекомендуем установить `uvloop` в ваше окружение и настроить `loop = uvloop` в разделе `[uvicorn]` для повышения производительности.

[45] http://uwsgi-docs.readthedocs.io/en/latest/Configuration.html

3 Руководство по серверному API

Фреймворк VST Utils использует Django, Django Rest Framework, drf-yasg и Celery.

3.1 Модели

Модель - это единственный и окончательный источник истины о ваших данных. Она содержит основные поля и поведение для данных, которые вы храните. Хорошей практикой считается избегать написания собственных view и сериализаторов, поскольку BModel предоставляет богатый набор мета-атрибутов для их автоматической генерации в большинстве ситуаций.

Переопределение стандартных классов моделей Django в модуле *vstutils.models*.

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

Стандартный класс модели, генерирующий viewset, отдельные сериализаторы для list() и retrieve(), фильтры, api endpoint-ы и вложенные view.

Примеры:

```
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',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            '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
            }
        }
```

В данном случае создаются модели, которые затем будут конвертированы во view, где:

- `POST`/`GET` на `/api/version/task/` - создает новую задачу или получает список всех задач
- `PUT`/`PATCH`/`GET`/`DELETE` на `/api/version/task/:id/` - обновляет, получает или удаляет экземпляр задачи
- `POST`/`GET` to `/api/version/task/:id/stage/` - создает новую стадию или получает список всех стадий в задаче
- `PUT`/`PATCH`/`GET`/`DELETE` на `/api/version/task/:id/stage/:stage_id` - обновляет, получает или удаляет экземпляр стадии в задаче.

Чтобы добавить view к API, вставьте следующий код в *settings.py*:

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

Для первичного доступа к сгенерированному view, наследуйтесь от свойства *Task.generated_view*.

Чтобы упростить процесс перевода на фронтенде, используйте атрибут `_translate_model` вместе с названием модели.

Список мета-атрибутов для генерации view:

- `_view_class` - список дополнительных классов view для наследования. Класс, унаследованный от ViewSet или строка для его импорта. Константы также поддерживаются:
  - `read_only` - для создания view, поддерживающего только просмотр;
  - `list_only` - для создания view, поддерживающего только список;
  - `history` - для создания view, поддерживающего только просмотр и удаление записей.

Представление, поддерживающее все CRUD-операции, применяется по умолчанию.

- `_serializer_class` - класс API сериализатора; используйте этот атрибут, чтобы указать родительский класс автоматически сгенерированных сериализаторов. По умолчанию используется *vstutils.api.serializers.VSTSerializer*. Принимает строку для импорта, класс сериализатора или `django.utils.functional.SimpleLazyObject`.
- `_serializer_class_name` - название модели для OpenAPI definitions. Это название будет в сгенерированном интерфейсе администратора. По умолчанию используется имя класса.
- `_list_fields` или `_detail_fields` - список полей, которые будут отображены в списке или детальной записи соответственно. То же, что и мета-атрибут «fields» в сериализаторах DRF.
- `_override_list_fields` или `_override_detail_fields` - сопоставление имен и типов полей, которые будут переопределены в атрибутах сериализатора (думайте об этом как о переопределении полей в ModelSerializer из DRF).
- `_properties_groups` - словарь, где ключами являются названия групп, а значениями - списки полей (строки). Позволяет группировать поля в секции на фронтенде.
- `_view_field_name` - поле, которое будет использовано для вывода заголовка детальной записи.
- `_non_bulk_methods` - список методов, которые не должны использовать bulk для запросов.
- `_extra_serializer_classes` - сопоставление с дополнительными сериализаторами во viewset. Это может быть, например, сериализатор, который будет вычислять что-то в действии (имя сопоставления). Значением может быть строка для импорта. Важное замечание: при установке атрибута *model* в значение None будет использован стандартный механизм генерации сериализаторов, что позволит получить поля из list или detail сериализаторов (установите мета-атрибут сериализатора *__inject_from__* в *list* или *detail* соответственно). В некоторых случаях необходимо передать модель в сериализатор. Для этих целей используйте константу *LAZY_MODEL* в качестве мета-атрибута. Каждый раз, когда сериализатор будет использован, конкретная модель, в которой он объявлен, будет подставлена.
- `_filterset_fields` - список или словарь имен filterset для API-фильтрации. По умолчанию используется список полей list-view. При обработке списка полей проверяет наличие специальных имен полей и наследует дополнительные родительские классы. Если в списке есть `id`, класс будет наследован от *vstutils.api.filters.DefaultIDFilter*. Если есть `name` - от *vstutils.api.filters.DefaultNameFilter*. Если есть и `id`, и `name`, то класс будет наследован от обоих. Возможные значения включают *list* полей, которые нужно фильтровать, или *dict*, где ключ - имя поля, а значение - класс Filter. Использование словаря расширяет функциональность атрибута и дает возможность переопределить класс фильтра для отдельных полей (значение None выключает переопределение).
- `_search_fields` - кортеж или список полей, которые должны использоваться в поисковых запросах. По умолчанию (или если установлено *None*) - все фильтруемые поля в detail view.
- `_copy_attrs` - список полей экземпляра модели, указывающий, что экземпляр может быть скопирован с этими атрибутами.
- `_nested` - сопоставление ключ-значение вложенных view (ключ - имя вложенного view, kwargs для декоратора *vstutils.api.decorators.nested_view*, но поддерживает атрибут `model` в качестве вложенного). `model` может быть строкой для импорта.
- `_extra_view_attributes` - сопоставление ключ-значение дополнительных атрибутов view, имеет меньший приоритет перед сгенерированными атрибутами.

Как правило, вы также можете добавить другие атрибуты для переопределения или расширения списка классов обработки по умолчанию. Поддерживаются `filter_backends`, `permission_classes`, `authentication_classes`, `throttle_classes`,

renderer_classes и parser_classes. Список мета-атрибутов для настройки выглядит так:

- _pre_{attribute} - Список классов, включаемых до классов по умолчанию.
- _{attribute} - Список классов, включаемых после классов по умолчанию.
- _override_{attribute} - булев флаг, указывающий, что атрибут переопределяет стандартный viewset (в противном случае расширяет). По умолчанию: False.

Примечание: Возможно, вам потребуется создать экшен[46] в сгенерированном view. Используйте декоратор *vstutils.models.decorators.register_view_action* с аргументом detail, чтобы применить его к списку или детальной записи. В этом случае декорированный метод будет принимать экземпляр view в self.

Примечание: В некоторых случаях, наследование модели может также требовать наследования класса Meta базовой модели. Если Meta явно объявлен в базовом классе, то вы можете получить его с помощью атрибута *OriginalMeta* и использовать его для наследования.

Примечание: Строка документации модели будет переиспользована для описания view. Есть возможность сделать общее описание для всех экшенов и описание для каждого отдельно, используя следующий синтаксис:

```
General description for all actions.

action_name:
    Description for this action.

another_action:
    Description for another action.
```

---

hidden

Если hidden установлено в True, вхождение будет исключено из запроса в BQuerySet.

id

Первичное поле для выборки и поиска в API.

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

Стандартный VSTUtils-менеджер. Используется классами *BaseModel* и *BModel*. Использует *BQuerySet* в качестве базового.

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

Представляет ленивый поиск в базе данных множества объектов. Позволяет перегрузить итерируемый класс по умолчанию с помощью атрибута *custom_iterable_class* (класс с методом *__iter__*, возвращающий генератор объектов модели) и стандартный класс запроса с помощью атрибута *custom_query_class* (дочерний класс django.db.models.sql.query.Query).

cleared()

Фильтрует queryset для моделей с атрибутом *hidden*, исключая все скрытые объекты.

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

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

Возвращает инициализированные объекты класса *vstutils.utils.Paginator* через текущий экземпляр QuerySet. Все аргументы и (args) и именованные аргументы (kwargs) попадают в конструктор класса Paginator.

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

Возвращает разбитые на страницы данные при помощи пользовательского класса Paginator. Используйте *PAGE_LIMIT* из глобальных настроек по умолчанию.

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

Декоратор, позволяющий обратить методы модели в сгенерированные экшены[47] view. Декорированный метод становится методом сгенерированного view, где *self* - объект view. Смотрите поддерживаемые аргументы в *vstutils.api.decorators.subaction()*.

Примечание: Возможно вам потребуется использовать прокси-модели со стандартным набором экшенов. Чтобы получить экшен прокси-модели, передайте именованный аргумент `inherit` со значением `True`.

Примечание: Часто экшен не должен передавать никаких параметров, отправляя вместо этого пустой запрос. Чтобы ускорить разработку, мы установили сериализатор *vstutils.api.serializers.EmptySerializer*. по умолчанию.

Вы также можете использовать модели, не нуждающиеся в базе данных:

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

Данная кастомная модель предназначена для самостоятельной реализации запросов к внешним сервисам. Модель позволяет вам передавать параметры фильтрации, лимитирования и сортировки к внешним запросам, получая уже ограниченные данные.

Чтобы начать использовать эту модель, достаточно реализовать метод класса `get_data_generator()`, который получает объект запроса с необходимыми параметрами, как аргумент.

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

Модель, загружающая данные из YAML-файла вместо базы данных. Путь до файла хранится в атрибуте *FileModel.file_path*.

Примеры:

Исходный файл хранится в */etc/authors.yaml* вместе со следующим содержимым:

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

Пример:

```
from vstutils.custom_model import FileModel, CharField


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

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

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

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

Модель, использующая список или словарь для хранения данных (атрибут *ListModel.data*) вместо базы данных. Полезна в том случае, если у вас простой набор данных.

Примеры:

```
from vstutils.custom_model import ListModel, CharField


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

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

Иногда может быть необходимо переключаться между источниками данных. Для этих целей следует использовать функцию *setup_custom_queryset_kwargs*, которая принимает именованные аргументы, отправляющиеся затем в функцию инициализации данных. Один из таких аргументов для *ListModel* - date_source, принимающий любой итерируемый объект.

Примеры:

```
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"},
])
```

В этом случае мы задаем список источников через функцию *setup_custom_queryset_kwargs*, и каждый последующий вызов в цепочке методов будет работать с этими данными.

data = []

Список кортежей данных. Пустой по умолчанию.

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

Эта модель реализует механизм программирования SQL View над другими моделями. В методе `get_view_queryset()` подготавливается запрос, и все остальные действия осуществляются поверх него.

3.1.1 Поля Модели

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*)

Расширяет `django.db.models.ForeignKey`[48]. Используйте это поле в *vstutils.models.BModel*, чтобы получить *vstutils.api.FkModelField* в сериализаторе. Чтобы установить Foreign Key отношение, задайте значение *to* - класс модели или строка для импорта, как в `django.db.models.ForeignKey`[49]

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

Расширяет класс `django.db.models.TextField`[50]. Простое поле для хранения HTML-разметки. Поле основано на базе `django.db.models.TextField`[51], поэтому не поддерживает индексацию и не рекомендовано для использования в фильтрах.

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

Подклассы `django.db.models.fields.files.FieldFile`[52]. Предоставляют *`MultipleFieldFile.save()`* и *`MultipleFieldFile.delete()`* для управления базовым файлом, а также для обновления соответствующего экземпляра модели.

delete(*save=True*)

Удаляет файл из хранилища и из атрибута объекта.

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

Сохраняет изменения в файле в хранилище и атрибут объекта.

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

Подклассы `django.db.models.fields.files.FileDescriptor` для обработки списка файлов. Возвращает список *`MultipleFieldFile`* при обращении, поэтому вы можете написать такой код:

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

get_file(*file*, *instance*)

Всегда возвращает валидный объект attr_class. За деталями реализации обратитесь к `django.db.models.fields.files.FileDescriptor.__get__()`.

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

Подклассы `django.db.models.fields.files.FileField`. Поле для хранения списка файлов, содержащихся в хранилище. Все аргументы передаются в FileField.

attr_class

alias of *`MultipleFieldFile`*

descriptor_class

alias of *`MultipleFileDescriptor`*

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

Миксина, предназначенная для использования вместе с `django.db.models.fields.files.FieldFile`[53] для преобразования его в Field вместе со списком файлов.

get_prep_value(*value*)

Подготовка значения для вставки в базу данных

pre_save(*model_instance*, *add*)

Вызов метода .save() для каждого файла списка

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

Поле для хранения списка изображения, содержащихся в хранилище. Все аргументы передаются в `django.db.models.fields.files.ImageField`, кроме height_field и width_field, так как они пока не реализованы.

[48] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.ForeignKey
[49] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.ForeignKey
[50] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[51] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[52] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.fields.files.FieldFile
[53] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.fields.files.FieldFile

attr_class

alias of *MultipleImageFieldFile*

descriptor_class

alias of *MultipleFileDescriptor*

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

Подклассы *MultipleFieldFile* и ImageFile mixin, обрабатывают удаление _dimensions_cache, когда файл удаляется.

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

Расширяет django.db.models.TextField[54]. Используйте это поле в *vstutils.models.BModel*, чтобы получить *vstutils.api.MultipleNamedBinaryFileInJSONField* в сериализаторе.

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

Расширяет django.db.models.TextField[55]. Используйте это поле в *vstutils.models.BModel*, чтобы получить *vstutils.api.MultipleNamedBinaryImageInJSONField* в сериализаторе.

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

Расширяет django.db.models.TextField[56]. Используйте это поле в *vstutils.models.BModel*, чтобы получить *vstutils.api.NamedBinaryFileInJSONField* в сериализаторе.

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

Расширяет django.db.models.TextField[57]. Используйте это поле в *vstutils.models.BModel*, чтобы получить *vstutils.api.NamedBinaryImageInJSONField* в сериализаторе.

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

Расширяет django.db.models.TextField[58]. Простое поле для хранения строк в формате Markdown. Поле основано на django.db.models.TextField[59], поэтому не поддерживает индексацию и не рекомендовано для использования в фильтрах.

3.2 Веб-API

Веб-API основано на Django Rest Framework. Предоставляет дополнительные вложенные функции.

[54] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[55] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[56] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[57] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[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

3.2.1 Поля

Фреймворк включает в себя список удобных полей сериализатора. Некоторые из них вступают в силу только в сгенерированном интерфейсе администратора.

Дополнительные поля сериализатора для генерации OpenAPI и графического интерфейса.

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

Поле, предоставляющее автодополнение на фронтенде. Использует указанный список объектов.

Параметры

- **autocomplete** (*list*[60], *tuple*[61], *str*[62]) – Источник для автодополнения. Можно задать список/кортеж со значениями или definition name в схеме OpenAPI. Для definition пользовательский интерфейс найдет оптимальную ссылку и отобразит значения, основанные на аргументах `autocomplete_property` и `autocomplete_represent`.
- **autocomplete_property** (*str*[63]) – этот аргумент указывает, какие атрибуты будут взяты из model definition схемы OpenAPI в качестве используемого значения.
- **autocomplete_represent** – этот аргумент указывает, какие атрибуты будут взяты из model definition схемы OpenAPI в качестве отображаемого значения.
- **use_prefetch** (*bool*[64]) – делает prefetch для значений на фронтенде в list-view. `True` по умолчанию.

Примечание: Действует только в графическом интерфейсе. Работает аналогично *VSTCharField* в API.

---

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

Простое строковое поле. Значение всегда должно быть допустимой строкой ASCII. В пользовательском интерфейсе оно будет отображено как штрихкод (код 128).

Параметры

child (*rest_framework.fields.Field*) – поле для сериализации или десериализации оригинальных данных.По умолчанию: `rest_framework.fields.CharField`

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

Поле, расширяющее *FileInStringField*, но работающее также с бинарными (base64) файлами.

Параметры

media_types (*tuple*[65], *list*[66]) – Список MIME-типов, доступных для выбора пользователем. Поддерживается синтаксис с использованием `*`. По умолчанию *[„*/*“]*

Примечание: Действует только в графическом интерфейсе. Работает аналогично *VSTCharField* в API.

---

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

Поле, расширяющее *FileInStringField*, используется для работы с csv файлами. Обеспечивает отображение загруженных данных в виде таблицы.

[60] https://docs.python.org/3.6/library/stdtypes.html#list
[61] https://docs.python.org/3.6/library/stdtypes.html#tuple
[62] https://docs.python.org/3.6/library/stdtypes.html#str
[63] https://docs.python.org/3.6/library/stdtypes.html#str
[64] https://docs.python.org/3.6/library/functions.html#bool
[65] https://docs.python.org/3.6/library/stdtypes.html#tuple
[66] https://docs.python.org/3.6/library/stdtypes.html#list

Параметры

- **items** (*Serializer*) – Конфигурация таблицы. Это сериализатор drf или vst, включающий CharField'ы, которые являются ключами словарей, и именами колонок в таблице. В ключи сериализуются данные из csv. Поля должны быть в том порядке, в котором вы хотите видеть их в таблице. Следующие опции могут также быть включены: - `label`: удобочитаемое название колонки - `required`: определяет, будет ли поле обязательным. По умолчанию False.
- **min_column_width** (*int*[67]) – Минимальная ширина ячейки. По умолчанию 200 px.
- **delimiter** (*str*[68]) – Символ-разделитель.
- **lineterminator** (*str*[69]) – Последовательность символов новой строки. Оставьте пустым для выбора автоматически. Возможные значения: `\r`, `\n`, `\r\n`.
- **quotechar** (*str*[70]) – Символ, используемый в качестве кавычек для полей.
- **escapechar** (*str*[71]) – Символ, используемый для экранирования кавычки в поле.
- **media_types** (*tuple*[72], *list*[73]) – Список MIME-типов, доступных для выбора пользователем. Поддерживается синтаксис с использованием `*`. По умолчанию `['text/csv']`

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

Поле, содержащее список значений с указанным разделителем (по умолчанию «,»). Получает список значений из другой модели или списка. Предоставляет автодополнение, как и *AutoCompletionField*, но со списками в виде строки, где слова разделяются запятыми. Подходит для полей-свойств модели, где уже реализована основная логика, или для поля `model.CharField`.

Параметры

- **select** (*str*[74], *tuple*[75], *list*[76]) – Определение имени схемы OpenAPI или списка со значениями.
- **select_separator** (*str*[77]) – разделитель значений. По умолчанию запятая.
- **select_property, select_represent** – работает так же, как `autocomplete_property` и `autocomplete_represent` По умолчанию `name`.
- **use_prefetch** – делает prefetch значений на фронтенде в list-view. По умолчанию `False`.
- **make_link** – Отображает значение как ссылку на модель. По умолчанию `True`.
- **dependence** (*dict*[78]) – Словарь, где ключи - это имена полей из той же модели, а значения - названия query-фильтров. Если хотя бы одно из полей, от которых существует зависимость не допускает null, обязательно или установлено в null, список автодополнения будет пустым, а поле окажется выключенным.

Примечание: Действует только в графическом интерфейсе. Работает аналогично *VSTCharField* в API.

---

[67] https://docs.python.org/3.6/library/functions.html#int
[68] https://docs.python.org/3.6/library/stdtypes.html#str
[69] https://docs.python.org/3.6/library/stdtypes.html#str
[70] https://docs.python.org/3.6/library/stdtypes.html#str
[71] https://docs.python.org/3.6/library/stdtypes.html#str
[72] https://docs.python.org/3.6/library/stdtypes.html#tuple
[73] https://docs.python.org/3.6/library/stdtypes.html#list

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

Простое поле, аналогичное crontab, содержащее расписание cron-записей для указания времени. Поле crontab имеет пять полей для указания дня, даты и времени. * в поле значений выше означает все допустимые значения, указанные в скобках для данного столбца.

В поле значений может быть * или список элементов, разделенных запятыми. Элементом может быть число из указанных выше диапазонов или два числа из диапазона, разделенных дефисом (означает включительный диапазон).

Поля времени и даты:

Поля	допустимое значение
minute	0-59
hour	0-23
day of month	1-31
month	1-12
day of week	0-7 (0 или 7 - Sunday)

Значение по умолчанию для каждого поля, если не указано, составляет

```
.---------------- 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*)

Расширяет *FkModelField*, но отображается в виде дерева на фронтенде.

Предупреждение: Это поле не поддерживает `dependence`. Используйте `filters` на свой страх и риск, так как они могут сломать структуру дерева.

Параметры

- **only_last_child** (*bool*[79]) – если True, то допускает к выбору только то значение, которое не имеет дочерних элементов. По умолчанию *False*
- **parent_field_name** (*str*[80]) – название родительского поля в модели. По умолчанию *parent*

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

Поле, расширяющее *DynamicJsonTypeField*, но его значение не преобразуется в json, а остается как есть. Полезно при использовании `property`[81] в модели или для экшенов.

Параметры

[74] https://docs.python.org/3.6/library/stdtypes.html#str
[75] https://docs.python.org/3.6/library/stdtypes.html#tuple
[76] https://docs.python.org/3.6/library/stdtypes.html#list
[77] https://docs.python.org/3.6/library/stdtypes.html#str
[78] https://docs.python.org/3.6/library/stdtypes.html#dict
[79] https://docs.python.org/3.6/library/functions.html#bool
[80] https://docs.python.org/3.6/library/stdtypes.html#str

- **field** (*str*[82]) – поле в модели, изменение значения которого будет менять тип текущего значения.
- **types** – сопоставление ключ-значение, где ключом является значение поля-подписчика, а значением - тип (формата OpenAPI) текущего поля.
- **choices** (*dict*[83]) – варианты выбора для разных значений подписанных полей. Использует сопоставление, где ключом является подписанное поле, а значением - список значений для выбора.

Примечание: Действует только в графическом интерфейсе. В API работает аналогично *VSTCharField* без изменения значения.

---

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

Поле, расширяющее *DynamicJsonTypeField*. Валидирует данные поля с помощью `field_attribute`, выбранного в связанной модели. По умолчанию любое значение `field_attribute` валидируется классом *VSTCharField*. Чтобы переопределить это поведение, установите словарный атрибут `{field_attribute value}_fields_mapping` в связанной модели, где:

- **key** - строковое представление типа значения, получаемое от связанной модели `field_attribute`.
- **value** - экземпляр `rest_framework.Field` для валидации.

Параметры

- **field** (*str*[84]) – поле в модели, чье изменение значения меняет тип текущего значения. Поле должно быть классом *FkModelField*.
- **field_attribute** (*str*[85]) – атрибут связанного экземпляра модели с именем типа.

Предупреждение: `field_attribute` в связанной модели должно быть типа `rest_framework.fields.ChoicesField`, иначе в графическом интерфейсе поле будет отображаться как обычное текстовое.

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

Поле, тип которого зависит от другого поля. Хранит значение в виде строки, а отображает поле в виде объекта json.

Параметры

- **field** (*str*[86]) – поле в модели, изменение значения которого будет менять тип текущего значения.
- **types** – сопоставление ключ-значение, где ключом является значение поля-подписчика, а значением - тип (формата OpenAPI) текущего поля.
- **choices** (*dict*[87]) – варианты выбора для разных значений подписанных полей. Использует сопоставление, где ключом является подписанное поле, а значением - список значений для выбора.

[81] https://docs.python.org/3.6/library/functions.html#property
[82] https://docs.python.org/3.6/library/stdtypes.html#str
[83] https://docs.python.org/3.6/library/stdtypes.html#dict
[84] https://docs.python.org/3.6/library/stdtypes.html#str
[85] https://docs.python.org/3.6/library/stdtypes.html#str

- **source_view** (*str*[88]) – Позволяет использовать данные родительских view в качестве источника для создания полей. Можно указать точный путь представления (*/user/{id}/*) или относительный спецификатор родительского представления (*<<parent>>.<<parent>>.<<parent>>*). Например, если текущая страница - */user/1/role/2/*, а *source_view* - *<<parent>>.<<parent>>*, то будут использованы данные из */user/1/*. Поддерживаются только представления деталей.

Примечание: Действует только в графическом интерфейсе. В API работает аналогично *VSTCharField* без изменения значения.

---

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

Поле, расширяющее *VSTCharField*. Сохраняет содержимое файла в виде строки.

Поле должно быть текстовым (не бинарным). Сохраняется в модель как есть.

Параметры

media_types (*tuple*[89], *list*[90]) – Список MIME-типов, доступных для выбора пользователем. Поддерживается синтаксис с использованием `*`. По умолчанию `['*/*']`

Примечание: Действует только в графическом интерфейсе. В API ведет себя так же, как и *VSTCharField*.

---

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

Реализация ForeignKeyField. Вы можете указать, какое поле связанной модели будет храниться в этом поле (по умолчанию: «id») и какое будет отображаться на фронтенде.

Параметры

- **select** (*str*[91]) – Имя определения схемы OpenAPI.
- **autocomplete_property** (*str*[92]) – этот аргумент указывает, какие атрибуты будут взяты из model definition схемы OpenAPI в качестве используемого значения. По умолчанию `id`.
- **autocomplete_represent** – этот аргумент указывает, какие атрибуты будут взяты из model definition схемы OpenAPI в качестве отображаемого значения. По умолчанию `name`.
- **field_type** (*type*[93]) – Определяет тип поля autocomplete_property для дальнейшего описания в схеме и преобразования этого типа из API. По умолчанию пропускается, но требуект объекты *int`или `str*.
- **use_prefetch** (*bool*[94]) – делает prefetch для значений на фронтенде в list-view. `True` по умолчанию.
- **make_link** (*bool*[95]) – Отображает значение как ссылку на модель. По умолчанию `True`.
- **dependence** (*dict*[96]) – словарь, где ключи - это имена полей из той же модели, а значения - названия query-фильтров. Если хотя бы одно из полей, от которых существует зависимость, не допускает null, обязательно или установлено в null, список автодополнения будет пустым, а поле окажется выключенным. Есть несколько специальных

[86] https://docs.python.org/3.6/library/stdtypes.html#str
[87] https://docs.python.org/3.6/library/stdtypes.html#dict
[88] https://docs.python.org/3.6/library/stdtypes.html#str
[89] https://docs.python.org/3.6/library/stdtypes.html#tuple
[90] https://docs.python.org/3.6/library/stdtypes.html#list

ключей dependence-словаря, с помощью которых можно получить данные, хранящиеся на фронтенде, не делая лишних запросов в базу данных: `'<<pk>>'` получает первичный ключ текущего экземпляра, `'<<view_name>>'` получает имя view из компонента Vue, `'<<parent_view_name>>'` получает имя родительского view из компонента Vue, `'<<view_level>>'` получает уровень view, `'<<operation_id>>'` получает operation_id, `'<<parent_operation_id'>>` получает родительский operation_id.

Примеры:

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

Этот фильтр получит первичный ключ текущего объекта и сделает запрос на фронтенде „/category?my_filter=3“, где „3“ - первичный ключ текущего экземпляра.

Параметры

filters (*dict*[97]) – словарь, где ключи - это имена поля связанной модели, а значения - значения этого поля.

Примечание: Пересечение *dependence.values()* и *filters.keys()* выкинет ошибку для предотвращения неоднозначности при фильтрации.

Примечание: Действует только в графическом интерфейсе. Работает аналогично `rest_framework.fields.IntegerField` в API.

---

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

Расширяет *FkField*, но хранит указанный класс модели. Это поле полезно для установки полей `django.db.models.ForeignKey`[98] в модели.

Параметры

- **select** (`vstutils.models.BModel`, `vstutils.api.serializers.VSTSerializer`) – класс модели (основанный на *vstutils.models.BModel*) или сериализатор, используемый в API и имеющий свой путь в схеме OpenAPI.
- **autocomplete_property** (*str*[99]) – этот аргумент указывает, какие атрибуты будут взяты из model definition схемы OpenAPI в качестве используемого значения. По умолчанию `id`.
- **autocomplete_represent** – этот аргумент указывает, какие атрибуты будут взяты из model definition схемы OpenAPI в качестве отображаемого значения. По умолчанию `name`.
- **use_prefetch** – делает prefetch для значений на фронтенде в list-view. `True` по умолчанию.
- **make_link** – Отображает значение как ссылку на модель. По умолчанию `True`.

[91] https://docs.python.org/3.6/library/stdtypes.html#str
[92] https://docs.python.org/3.6/library/stdtypes.html#str
[93] https://docs.python.org/3.6/library/functions.html#type
[94] https://docs.python.org/3.6/library/functions.html#bool
[95] https://docs.python.org/3.6/library/functions.html#bool
[96] https://docs.python.org/3.6/library/stdtypes.html#dict
[97] https://docs.python.org/3.6/library/stdtypes.html#dict

Предупреждение: Класс модели получает объект из базы данных в процессе выполнения *.to_internal_value*. Будьте осторожны при выполнении массовых сохранений.

Предупреждение: Permission'ы модели, на которую ссылается это поле, не проверяются. Следует их проверять вручную в сигналах или валидаторах.

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

Поле, содержащее текст html и помеченное как format:html. Это поле не проверяет, является ли его содержимое валидным HTML.

Предупреждение: Чтобы избежать уязвимости, не позволяйте пользователям изменять эти данные, так как они могут выполнить нежелательный скрипт.

Примечание: Действует только в графическом интерфейсе. Работает аналогично *VSTCharField* в API.

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

Расширяет класс „rest_framework.serializers.CharField". Поле, применяющее маску к значению.

Параметры

mask (*dict*[100], *str*[101]) – IMask[102]

Примечание: Действует только на фронтенде.

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

Расширяет *NamedBinaryFileInJsonField*, но использует список JSON'ов. Позволяет оперировать несколькими файлами через список объектов *NamedBinaryFileInJsonField*.

Атрибуты: `NamedBinaryInJsonField.file`: если True, принимает только подклассы File в качестве входных данных. Если False, принимает только значения типа string. По умолчанию: False.

file_field

alias of *MultipleFieldFile*

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

Расширяет *MultipleNamedBinaryFileInJsonField*, но использует список JSON'ов. Используется для оперирования несколькими изображениями и работает как список объектов *NamedBinaryImageInJsonField*.

Параметры

background_fill_color (*str*[103]) – Цвет для заполнения области, не покрытой изображением после обрезки. По умолчанию прозрачный, но будет черным, если формат изображения не поддерживает прозрачность. Может быть любым допустимым цветом CSS.

[98] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.ForeignKey
[99] https://docs.python.org/3.6/library/stdtypes.html#str
[100] https://docs.python.org/3.6/library/stdtypes.html#dict
[101] https://docs.python.org/3.6/library/stdtypes.html#str
[102] https://imask.js.org/guide.html
[103] https://docs.python.org/3.6/library/stdtypes.html#str

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

Поле, принимающее на вход JSON со следующими свойствами: * name - string - имя файла; * mediaType - string - MIME-тип файла; * content - base64 string - содержимое файла.

Это поле полезно для сохранения бинарных файлов с их именами в полях модели `django.db.models.CharField`[104] или `django.db.models.TextField`[105]. Все операции по кодированию или декодированию бинарного содержимого осуществляется на клиенте. Это накладывает разумные ограничения на размер файла.

Кроме того, это поле может создать `django.core.files.uploadedfile.SimpleUploadedFile` из входящего JSON и сохранить его как файл в `django.db.models.FileField`[106], если для аргумента *file* установлено значение *True*

Атрибуты: `NamedBinaryInJsonField.file`: если True, принимает только подклассы File в качестве входных данных. Если False, принимает только значения типа string. По умолчанию: False.

Примечание: Действует только в графическом интерфейсе. Работает аналогично *VSTCharField* в API.

---

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

Расширяет *NamedBinaryFileInJsonField* для view изображения на фронтенде (если бинарное изображение валидно). Валидируйте это поле с помощью *vstutils.api.validators.ImageValidator*.

Параметры

background_fill_color (*str*[107]) – Цвет для заполнения области, не покрытой изображением после обрезки. По умолчанию прозрачный, но будет черным, если формат изображения не поддерживает прозрачность. Может быть любым допустимым цветом CSS.

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

Расширяет CharField[108], но в схеме имеет format = *password*. В пользовательском интерфейсе отображает все символы как звездочки вместо реально введенных данных.

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

Расширяет класс „rest_framework.serializers.CharField“. Поле для ввода номера телефона в международном формате

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

Простое строковое поле. В пользовательском интерфейсе оно будет отображено как QR-код.

Параметры

child (*rest_framework.fields.Field*) – поле для сериализации или десериализации оригинальных данных.По умолчанию: `rest_framework.fields.CharField`

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

Расширяет класс „rest_framework.serializers.FloatField“. Это поле представляет собой ввод рейтинга пользователем на фронтенде. Пределы оценок могут быть заданы с помощью „min_value=“ и „max_value=“, по умолчанию 0 и 5 соответственно. Минимальный шаг между оценками определяется параметром „step=“, по умолчанию 1. Внешний вид на фронтенде может быть выбран с помощью „front_style=“, доступные варианты перечислены в „self.valid_front_styles“.

[104] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.CharField
[105] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[106] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.FileField
[107] https://docs.python.org/3.6/library/stdtypes.html#str
[108] https://www.django-rest-framework.org/api-guide/fields/#charfield

Для стиля „slider“ вы можете указать цвет слайдера, передав валидный цвет в „color=“. Для стиля „fa_icon“ вы можете указать иконку FontAwesome, которая будет использована для отображения рейтинга, передав валидный код иконки FontAwesome в „fa_class=“.

Параметры

- **min_value** (*float*[109], *int*[110]) – минимальный уровень
- **max_value** (*float*[111], *int*[112]) – максимальный уровень
- **step** (*float*[113], *int*[114]) – минимальный шаг между уровнями
- **front_style** (*str*[115]) – визуализация поля на фронтенде. Допустимы: [„stars“, „slider“, „fa_icon“].
- **color** (*str*[116]) – цвет элемента рейтинга (star, icon или slider) в формате css
- **fa_class** (*str*[117]) – код иконки FontAwesome

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

Поле для редиректа по строке. Часто используется в экшенах для редиректа после выполнения.

Примечание: Действует только в графическом интерфейсе. Работает аналогично `rest_framework.fields.IntegerField` в API.

---

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

Миксина поля, указывающая, что это поле используется для отправки адреса редиректа после некоторого действия.

Параметры

- **operation_name** (*str*[118]) – префикс для operation_id, например, если operation_id = *history_get*, то operation_name = *history*
- **depend_field** (*str*[119]) – имя поля, от которого оно зависит, его значение будет использовано для operation_id
- **concat_field_name** (*bool*[120]) – если True, то имя поля будет добавлено в конец operation_id

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

Поля для редиректа по id. Часто используется в экшенах для редиректа после выполнения.

Примечание: Действует только в графическом интерфейсе. Работает аналогично `rest_framework.fields.IntegerField` в API.

---

[109] https://docs.python.org/3.6/library/functions.html#float
[110] https://docs.python.org/3.6/library/functions.html#int
[111] https://docs.python.org/3.6/library/functions.html#float
[112] https://docs.python.org/3.6/library/functions.html#int
[113] https://docs.python.org/3.6/library/functions.html#float
[114] https://docs.python.org/3.6/library/functions.html#int
[115] https://docs.python.org/3.6/library/stdtypes.html#str
[116] https://docs.python.org/3.6/library/stdtypes.html#str
[117] https://docs.python.org/3.6/library/stdtypes.html#str
[118] https://docs.python.org/3.6/library/stdtypes.html#str
[119] https://docs.python.org/3.6/library/stdtypes.html#str
[120] https://docs.python.org/3.6/library/functions.html#bool

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

Расширяет класс *VSTCharField*. С этим полем вы можете получить обратное ForeignKey отношение как список связанных экземпляров.

Для использования следует указать kwarg „related_name“ (related_manager для обратного ForeignKey) и kwarg „fields“ (список или кортеж полей из связанной модели, которая должна быть включена).

По умолчанию *VSTCharField* используется для сериализации всех значений поля и отображения их на фронтенде. Вы можете указать *serializer_class* и переопределить поля, которые вам нужны. Например, title, description или другие поля, свойства которых можно задать так, чтобы определить новое поведение на фронтенде.

Параметры

- **related_name** (*str*[121]) – имя related manager'а для обратного foreign key отношения
- **fields** (*list*[122]*[str*[123]*]*, *tuple*[124]*[str*[125]*]*) – список связанных полей модели.
- **view_type** (*str*[126]) – определяет, как поля будут отображены на фронтенде. Должен быть либо „list“, либо „table“.
- **fields_custom_handlers_mapping** (*dict*[127]) – включает пользовательские handler'ы, где ключ: field_name, значение: callable_obj, который принимает параметры: instance[dict], fields_mapping[dict], model, field_name[str]
- **serializer_class** (*type*[128]) – Сериализатор для определения типов полей, если не задано, будет использован *VSTCharField* для каждого поля из списка *fields*

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

Поле, расширяющее *FileInStringField*, но скрывающее свое значение в интерфейсе администратора.

Поле должно быть текстовым (не бинарным). Сохраняется в модель как есть.

Параметры

media_types (*tuple*[129]*,list*[130]) – Список MIME-типов, доступных для выбора пользователем. Поддерживается синтаксис с использованием `*`. По умолчанию `['*/*']`

Примечание: Действует только в графическом интерфейсе. В API ведет себя так же, как и *VSTCharField*.

---

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

Поле, содержащее многострочный текст.

Примечание: Действует только в графическом интерфейсе. Работает аналогично *VSTCharField* в API.

---

[121] https://docs.python.org/3.6/library/stdtypes.html#str
[122] https://docs.python.org/3.6/library/stdtypes.html#list
[123] https://docs.python.org/3.6/library/stdtypes.html#str
[124] https://docs.python.org/3.6/library/stdtypes.html#tuple
[125] https://docs.python.org/3.6/library/stdtypes.html#str
[126] https://docs.python.org/3.6/library/stdtypes.html#str
[127] https://docs.python.org/3.6/library/stdtypes.html#dict
[128] https://docs.python.org/3.6/library/functions.html#type
[129] https://docs.python.org/3.6/library/stdtypes.html#tuple
[130] https://docs.python.org/3.6/library/stdtypes.html#list

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

Поле продолжительности времени, в секундах. Может быть использовано для подсчета времени работы чего-либо.

Примечание: Действует только в графическом интерфейсе. Работает аналогично `rest_framework.fields.IntegerField` в API.

---

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

CharField (расширяет `rest_framework.fields.CharField`). Это поле преобразует любой json в строку для модели.

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

На фронтенде отображается https://ui.toast.com/tui-editor. Сохраняет данные в формате markdown, экранируя все html-теги.

Параметры

escape (*bool*[131]) – экранирование входящих html-символов. Включено по умолчанию.

3.2.2 Валидаторы

Классы для валидации полей.

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

Базовый класс для валидации изображений. Проверяет media types.

Параметры

extensions (typing.Union[132][typing.Tuple[133], typing.List[134], None[135]]) – Кортеж или список файловых расширений, которые должны проходить проверку.

Выбрасывает rest_framework.exceptions.ValidationError в случае, если расширение файла не присутствует в списке

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

Проверяет размер изображения. Чтобы использовать этот класс для валидации ширины или высоты, переопределите self.orientation в („height“,), („width“,) или („height“, „width“)

Выбрасывает rest_framework.exceptions.ValidationError, если not(min <= (height or width) <= max)

Параметры

extensions (typing.Union[136][typing.Tuple[137], typing.List[138], None[139]]) –

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

Обертка для ImageBaseSizeValidator, проверяющая только высоту

Параметры

- **min_height** – минимальная валидная высота изображения

[131] https://docs.python.org/3.6/library/functions.html#bool
[132] https://docs.python.org/3.6/library/typing.html#typing.Union
[133] https://docs.python.org/3.6/library/typing.html#typing.Tuple
[134] https://docs.python.org/3.6/library/typing.html#typing.List
[135] https://docs.python.org/3.6/library/constants.html#None
[136] https://docs.python.org/3.6/library/typing.html#typing.Union
[137] https://docs.python.org/3.6/library/typing.html#typing.Tuple
[138] https://docs.python.org/3.6/library/typing.html#typing.List
[139] https://docs.python.org/3.6/library/constants.html#None

- **max_height** – максимальная валидная высота изображения
- **extensions** (typing.Union[140][typing.Tuple[141], typing.List[142], None[143]]) –

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

Валидатор изображения, который проверяет, может ли изображения быть распаковано из b64 в объект PIL Image. Не будет работать, если не установлен Pillow.

Выбрасывает rest_framework.exceptions.ValidationError, если PIL выбрасывает ошибку при попытке открыть изображение

Параметры

extensions (typing.Union[144][typing.Tuple[145], typing.List[146], None[147]]) –

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

Обертка для ImageBaseSizeValidator, проверяющая как высоту, так и ширину.

Параметры

- **min_height** – минимальная валидная высота изображения
- **max_height** – максимальная валидная высота изображения
- **min_width** – минимальная валидная ширина изображения
- **max_width** – максимальная валидная ширина изображения
- **extensions** (typing.Union[148][typing.Tuple[149], typing.List[150], None[151]]) –

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

Базовый класс для валидации изображения. Проверяет формат изображения. Не будет работать, если Pillow не установлен. Базовый класс для валидации изображения. Проверяет media types.

Параметры

extensions (typing.Union[152][typing.Tuple[153], typing.List[154], None[155]]) – Кортеж или список файловых расширений, которые должны проходить проверку.

Выбрасывает rest_framework.exceptions.ValidationError в случае, если расширение файла не присутствует в списке

property has_pillow

Проверьте, установлен ли Pillow

[140] https://docs.python.org/3.6/library/typing.html#typing.Union
[141] https://docs.python.org/3.6/library/typing.html#typing.Tuple
[142] https://docs.python.org/3.6/library/typing.html#typing.List
[143] https://docs.python.org/3.6/library/constants.html#None
[144] https://docs.python.org/3.6/library/typing.html#typing.Union
[145] https://docs.python.org/3.6/library/typing.html#typing.Tuple
[146] https://docs.python.org/3.6/library/typing.html#typing.List
[147] https://docs.python.org/3.6/library/constants.html#None
[148] https://docs.python.org/3.6/library/typing.html#typing.Union
[149] https://docs.python.org/3.6/library/typing.html#typing.Tuple
[150] https://docs.python.org/3.6/library/typing.html#typing.List
[151] https://docs.python.org/3.6/library/constants.html#None
[152] https://docs.python.org/3.6/library/typing.html#typing.Union
[153] https://docs.python.org/3.6/library/typing.html#typing.Tuple
[154] https://docs.python.org/3.6/library/typing.html#typing.List
[155] https://docs.python.org/3.6/library/constants.html#None

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

Обертка для ImageBaseSizeValidator, проверяющая только ширину

Параметры

- **min_width** – минимальная валидная ширина изображения
- **max_width** – максимальная валидная ширина изображения
- **extensions** (typing.Union[156][typing.Tuple[157], typing.List[158], None[159]]) –

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

Класс для валидации на основе регулярного выражения

Исключение
rest_framework.exceptions.ValidationError – в случае, если значение не соответствует регулярному выражению

Параметры
regexp (typing.Optional[160][typing.Pattern[161]]) –

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

Класс для валидации строки url query, например a=&b=1

Параметры
regexp (typing.Optional[162][typing.Pattern[163]]) –

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

Вспомогательная функция для изменения размера изображения пропорционально определенным значениям. Может создать белые поля в случае, если это необходимо для удовлетворения требуемого размера

Параметры

- **img** (*PIL.Image*) – объект Pillow Image
- **width** (*int*[164]) – Необходимая ширина
- **height** (*int*[165]) – Необходимая высота

Результат
объект Pillow Image

Тип результата
PIL.Image

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

Вспомогательная функция для изменения размера изображения пропорционально значениям между минимальным и максимальным значениями для каждой стороны. Может создать белые поля, если это необходимо для соблюдения ограничений

Параметры

[156] https://docs.python.org/3.6/library/typing.html#typing.Union
[157] https://docs.python.org/3.6/library/typing.html#typing.Tuple
[158] https://docs.python.org/3.6/library/typing.html#typing.List
[159] https://docs.python.org/3.6/library/constants.html#None
[160] https://docs.python.org/3.6/library/typing.html#typing.Optional
[161] https://docs.python.org/3.6/library/typing.html#typing.Pattern
[162] https://docs.python.org/3.6/library/typing.html#typing.Optional
[163] https://docs.python.org/3.6/library/typing.html#typing.Pattern
[164] https://docs.python.org/3.6/library/functions.html#int
[165] https://docs.python.org/3.6/library/functions.html#int

- **img** (*PIL.Image*) – объект Pillow Image
- **limits** (*dict*[166]) – Словарь с максимальным/минимальным ограничениями, например `{'width': {'min': 300, 'max: 600'}, 'height': {'min': 400, 'max: 800'}}`

Результат
объект Pillow Image

Тип результата
PIL.Image

3.2.3 Сериализаторы

Стандартные классы сериализаторов для web-api. Читайте подробнее в документации сериализаторов Django REST Framework Serializers[167].

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

Стандартный сериализатор с логикой работы с объектами. Читайте подробнее в документации сериализатора[168] как создавать сериализаторы и работать с ними.

Примечание: Вы также можете настроить `generated_fields` в атрибуте класса `Meta`, чтобы получить сериализатор с полями CharField по умолчанию. Настройте атрибут `generated_field_factory` чтобы изменить фабричный метод по умолчанию.

---

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

Стандартный сериализатор для пустых ответов. В сгенерированном графическом интерфейсе это означает, что кнопка действия не будет отображать дополнительного view перед запуском.

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

Стандартный сериализатор модели, основанный на `rest_framework.serializers.ModelSerializer`. Читайте подробнее в документации DRF[169], как создавать сериализаторы модели. Этот сериализатор сопоставляет полям модели расширенный набор полей сериализатора. Список доступных пар описан в *VSTSerializer.serializer_field_mapping*. Например, чтобы использовать *vstutils.api.fields.FkModelField* в сериализаторе, задайте *vstutils.models.fields.FkModelField* в модели.

3.2.4 Представления

Стандартные ViewSet'ы для web-api.

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

Миксина для viewset'ов, добавляющая *copy* endpoint во view.

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

Endpoint, который копирует экземпляр с его зависимостями.

copy_field_name = 'name'

Имя поля, которое получит префикс.

[166] https://docs.python.org/3.6/library/stdtypes.html#dict
[167] https://www.django-rest-framework.org/api-guide/serializers/
[168] https://www.django-rest-framework.org/api-guide/serializers/#serializers
[169] https://www.django-rest-framework.org/api-guide/serializers/#modelserializer

copy_prefix = 'copy-'

Значение префикса, которое будет добавлено к новому имени экземпляра.

copy_related = ()

Список связанных имен, которые будут скопированы в новый экземпляр.

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

Миксина ViewSet для получения FileResponse из моделей с файловыми полями.

Пример:

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

serializer_class_retrieve

alias of FileResponse

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

Базовый класс для всех view. Расширяет стандартные функции классов DRF. Здесь представлены некоторые из возможностей:

- Предоставляет атрибуты `model` вместо `queryset`.
- Позволяет устанавливать сериализаторы отдельно для каждого экшена через словарь `action_serializers` или атрибуты, имя которых соответствует шаблону `serializer_class_[action name]`.
- Позволяет отдельно указать сериализаторы для view списков и детальной записи.
- Оптимизирует запросы в базу данных для GET-запросов, делая выборку, если возможно, только тех полей, которые нужны сериализатору.

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

Метод, реализующий стандартную логику экшенов. Он опирается на переданные аргументы для построения логики. Поэтому, если был передан именованный аргумент, сериализатор будет подвержен валидации и сохранен.

Параметры

- **autosave** (*bool*[170]) – Включает/выключает выполнение сохранения сериализатором, если передан именованный аргумент *data*. Включено по умолчанию.
- **custom_data** (*dict*[171]) – Словарь с данными, которые будут переданы в *validated_data* без валидации.
- **serializer_class** (*None*, *type*[172]*[rest_framework.serializers.Serializer]*) – Класс сериализатора для текущего выполнения. Может быть полезно, когда сериализаторы запроса и ответа различны.

параметр
data: Стандартный аргумент класса сериализатора с сериализуемыми данными. Включает в себя валидацию и сохранение.

параметр
instance: Стандартный аргумент класса сериализатора с сериализуемым экземпляром.

Результат
Готовый сериализатор с логикой выполнения по умолчанию.

Тип результата
rest_framework.serializers.Serializer

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

Позволяет получить данные запроса и сериализовать значения, если существует атрибут *query_serializer_class* или атрибут был передан.

Параметры

- **request** – объект DRF запроса.
- **query_serializer** – класс сериализатора, для обработки параметров в query_params.
- **raise_exception** – флаг, который говорит о том нужно ли выбросить исключение при валидации в сериализаторе или нет.

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

Возвращает экземпляр сериализатора, который должен быть использован для валидации и десериализации входных данных, и сериализации выходных данных.

Позволяет использовать `django.http.StreamingHttpResponse`[173] в качестве инициализации сериализатора.

get_serializer_class()

Позволяет задать класс сериализатора для каждого экшена.

nested_allow_check()

Просто выбросьте исключение или пропустите (pass). Используется во вложенных view для упрощения проверки доступа.

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

Стандартный viewset, как, например ReadOnlyModelViewSet, но для данных исторического характера (позволяет удалять записи, но не создавать или обновлять). Наследуется от *GenericViewSet*.

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

Viewset, предоставляющий CRUD-экшены над моделью. Наследуется от *GenericViewSet*.

Переменные

- **model** (vstutils.models.BModel) – Модель БД с данными.
- **serializer_class** (vstutils.api.serializers.VSTSerializer) – Сериализатор для view данных модели.
- **serializer_class_one** (vstutils.api.serializers.VSTSerializer) – Сериализатор для view одного экземпляра данных модели.
- **serializer_class_[ACTION_NAME]** (vstutils.api.serializers.VSTSerializer) – Сериализатор для view любого endpoint'а, например *.create*.

Примеры:

```
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...
```

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

Стандартный viewset, как, например vstutils.api.base.ModelViewSet для readonly-моделей. Наследуется от *GenericViewSet*.

[170] https://docs.python.org/3.6/library/functions.html#bool
[171] https://docs.python.org/3.6/library/stdtypes.html#dict
[172] https://docs.python.org/3.6/library/functions.html#type
[173] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.StreamingHttpResponse

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

По умолчанию DRF не поддерживает вложенные view. Данный декоратор решает эту проблему.

Вам нужны две или более модели с вложенными отношениями (многие-ко-многим или многие-к-одному) и два viewset'а. Декоратор вкладывает viewset'ы в родительский класс viewset'ов и генерирует пути в API.

Параметры

- **name** (*str*[174]) – Имя вложенного пути. Также используется стандартное имя для связанных queryset'ов (см. *manager_name*).
- **arg** (*str*[175]) – Имя вложенного поля первичного ключа.
- **view** (vstutils.api.base.ModelViewSet, vstutils.api.base.HistoryModelViewSet, vstutils.api.base.ReadOnlyModelViewSet) – Класс вложенного viewset'а.
- **allow_append** (*bool*[176]) – Флаг, разрешающий добавление существующих экземпляров.
- **manager_name** (*str*[177], *Callable*[178]) – Имя атрибута объекта модели, который содержит вложенный queryset.
- **methods** (*list*[179]) – Список разрешенных методов для endpoint'ов вложенных view.
- **subs** (*list*[180], *None*) – Список разрешенных subviews или экшенов для endpoint'ов вложенных views.
- **queryset_filters** – Список вызываемых объектов, которые возвращают отфильтрованный queryset.

Примечание: Некоторые методы view не будут вызваны из соображений производительности. Это также применяется к некоторым атрибутам класса, которые обычно инициализируются в методах. Например, `.initial()` никогда не будет вызван. Каждый viewset обернут вложенным классом с дополнительной логикой.

Пример:

```
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
```

Данный код генерирует пути api:

- */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*)

Декоратор, оборачивающий метод объекта в subaction viewset'а.

Параметры

- **methods** – Список разрешенных методов HTTP. По умолчанию `["post"]`.
- **detail** – Флаг, указывающий, должен ли метод применяться к одному экземпляру.
- **serializer_class** – Сериализатор для этого экшена.
- **permission_classes** – Кортеж или список permission-классов.
- **url_path** – Имя API-пути для этого экшена.
- **description** – Описание этого экшена в OpenAPI.
- **multiaction** – Разрешает использовать этот экшен в мультиэкшенах. Работает только с *vstutils.api.serializers.EmptySerializer* в response.
- **require_confirmation** – Задает, должен ли экшен требовать подтверждения перед выполнением.
- **is_list** – Отметить это действие как пагинируемый список со всеми правилами и параметрами.
- **title** – Заменить заголовок действия.
- **icons** – Настроить классы иконок действия.

3.2.5 Actions (Действия)

Vstutils имеет развитую систему работы с действиями. REST API работает с данными через глаголы, которые называются методами. Однако для работы с одной или списком сущностей этих действий может быть недостаточно.

Чтобы расширить набор действий, необходимо создать действие, которое будет работать с некоторым аспектом описанной модели. Для этих целей существует стандартный `rest_framework.decorators.action()`, который также можно расширить с помощью схемы. Но для большего удобства в vstutils есть набор декораторов, которые позволяют избежать написания

Основная философия этих оберток заключается в том, что разработчик пишет бизнес-логику, не отвлекаясь на написание повторяющегося кода. Часто большинство ошибок в коде возникают именно из-за расфокусировки внимания при рутинном написании кода.

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*, ***kwargs*)

Базовый класс действий. Обладает минимально необходимой функциональностью для создания действия

[174] https://docs.python.org/3.6/library/stdtypes.html#str
[175] https://docs.python.org/3.6/library/stdtypes.html#str
[176] https://docs.python.org/3.6/library/functions.html#bool
[177] https://docs.python.org/3.6/library/stdtypes.html#str
[178] https://docs.python.org/3.6/library/typing.html#typing.Callable
[179] https://docs.python.org/3.6/library/stdtypes.html#list
[180] https://docs.python.org/3.6/library/stdtypes.html#list

и позволяет написать только бизнес-логику. Этот декоратор подходит в случаях, когда невозможно реализовать логику с использованием *SimpleAction* или алгоритм значительно более сложный, чем стандартные операции CRUD.

Примеры:

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

Параметры

- **detail** – Флаг, указывающий, какой тип действия используется: на списке или на отдельной сущности. Влияет на то, где будет отображаться это действие - на детальной записи или на списке записей.
- **methods** – Список доступных методов HTTP. По умолчанию `["post"]`.
- **serializer_class** – Сериализатор для тела запроса. Используется также для формирования ответа по умолчанию.
- **result_serializer_class** – Сериализатор для тела ответа. Необходим, когда запрос и ответ имеют различные наборы полей.
- **query_serializer** – Сериализатор для данных запроса типа GET. Используется, когда необходимо получить корректные данные в строке запроса типа GET и привести их к нужному типу.
- **multi** – Используется только с не-GET запросами и уведомляет GUI, что это действие должно быть применено к выбранным элементам списка.
- **title** – Заголовок действия в пользовательском интерфейсе. Для действий, отличных от GET, заголовок генерируется на основе имени метода.
- **icons** – Список иконок для кнопки пользовательского интерфейса.

- **is_list** – Флаг, указывающий, является ли тип действия списком или отдельной сущностью. Обычно используется с действиями GET.
- **kwargs** – Набор именованных аргументов для `rest_framework.decorators.action()`.

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

В этом случае действия с объектом не требуют каких-либо данных и манипуляций с ними. Для таких случаев существует стандартный метод, который позволяет упростить схему и код работы только с объектом.

При необходимости вы также можете переопределить сериализатор ответа, чтобы уведомить интерфейс о формате возвращаемых данных.

Примеры:

```
...
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 = ...
    ...

    @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.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
↪pdf'
        )
```

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

Идея этого декоратора заключается в том, чтобы получить полный CRUD для экземпляра с минимумом шагов. Экземпляр - это объект, который возвращается из декорируемого метода. Весь механизм очень похож на стандартный декоратор property, с описанием getter, setter, и deleter

Если вы собираетесь создать точку входа для работы с отдельным объектом, то вам не нужно определять методы. Наличие getter`a, setter`a, и deleter`a определит, какие методы будут доступны.

В официальной документации Django приведен пример с перемещением данных, которые не являются важными для авторизации, в модель Profile. Для работы с такими данными, находящимися вне основной модели, существует данный объект действия, который реализует основную логику в наиболеавтоматизированном виде.

Он охватывает большинство задач по работе с такими данными. По умолчанию у него есть метод GET вместо POST. Кроме того, для лучшей организации кода он позволяет изменить методы, которые будут вызываться при изменении или удалении данных.

При назначении действия на объект список методов также заполняется необходимыми методами.

Примеры:

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

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 Filterset'ы

Для большего удобства разработки, фреймворк предоставляет дополнительные классы и функции для фильтрации элементов на основе полей.

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

Базовый filterset для поиска по id. Предоставляет поиск по множеству значений, разделенных запятой. Использует *extra_filter()* в полях.

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

Базовый filterset для частичного поиска по названию. Использует условие *LIKE* в базе данных с помощью *name_filter()*.

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

Простой handler для фильтрации по связанным полям.

Параметры

- **related_pk** (str[181]) – Имя поля первичного ключа в связанной модели. По умолчанию „id".
- **related_name** (str[182]) – Имя charfield-поля в связанной модели. По умолчанию „name".
- **pk_handler** (typing.Callable[183]) – Изменяет handler для проверки значения перед поиском. Посылает «0», если handler падает. По умолчанию используется *int()*.

Пример:

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

Где author - это ForeignKey на *User*, и вы хотите искать по первичному ключу и полю email.

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

Метод, предназначенный для поиска значений в списке значений, разделенных запятой.

Параметры

- **queryset** (*django.db.models.query.QuerySet*[184]) – queryset модели для фильтрации.
- **field** (*str*[185]) – имя поля в FilterSet'е. Также поддерживает суффикс *__not*.
- **value** (*str*[186]) – список искомых значений, разделенных запятыми.

Результат
отфильтрованный queryset.

Тип результата
django.db.models.query.QuerySet[187]

[181] https://docs.python.org/3.6/library/stdtypes.html#str
[182] https://docs.python.org/3.6/library/stdtypes.html#str
[183] https://docs.python.org/3.6/library/typing.html#typing.Callable

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

Метод для частичного поиска по названию. Использует условие *LIKE* базы данных или выражение *contains* queryset'а.

Параметры

- **queryset** (*django.db.models.query.QuerySet*[188]) – queryset модели для фильтрации.
- **field** (*str*[189]) – имя поля в FilterSet'е. Также поддерживает суффикс *__not*.
- **value** (*str*[190]) – часть названия для поиска.

Результат

отфильтрованный queryset.

Тип результата

django.db.models.query.QuerySet[191]

3.2.7 Ответы (responses)

DRF предоставляет стандартный набор переменных, удобочитаемые названия которых соответствуют HTTP-кодам ответов. Для удобства мы динамически оборачиваем их в набор классов с соответствующими именами и дополнительно обеспечиваем следующие возможности:

- Строковые ответы оборачиваются в json, например `{ "detail": "string response" }`.
- Тайминги атрибутов сохраняются для дальнейшей обработки в middleware.
- Код состояния задается именем класса (например `HTTP_200_OK` или `Response200` имеют код 200).

Все классы наследуются от:

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

Класс ответа API со стандартным кодом состояния.

Переменные

- **status_code** (*int*[192]) – Код состояния HTTP.
- **timings** (*int*[193], *None*) – Тайминги ответов.

Параметры

timings – Тайминги ответов.

[184] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet
[185] https://docs.python.org/3.6/library/stdtypes.html#str
[186] https://docs.python.org/3.6/library/stdtypes.html#str
[187] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet
[188] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet
[189] https://docs.python.org/3.6/library/stdtypes.html#str
[190] https://docs.python.org/3.6/library/stdtypes.html#str
[191] https://docs.djangoproject.com/en/4.2/ref/models/querysets/#django.db.models.query.QuerySet
[192] https://docs.python.org/3.6/library/functions.html#int
[193] https://docs.python.org/3.6/library/functions.html#int

3.2.8 Middleware

По умолчанию Django предполагает[194], что разработчик создает класс Middleware вручную, однако зачастую это рутинная задача. Библиотека vstutils предлагает удобный класс request handler'а для изящной разработки в стиле ООП. Middleware используются для обработки входящих запросов и отправления ответов до того, как они достигнут получателя.

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

Базовый класс middleware для обработки:

- Входящие запросы от *BaseMiddleware.request_handler()*;
- Исходящий ответ перед любым обращением к серверу от *BaseMiddleware.get_response_handler()*;
- Исходящие ответы от *BaseMiddleware.handler()*.

Middleware должен быть добавлен в список *MIDDLEWARE*, находящийся в настройках.

Пример:

```
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
```

get_response_handler(*request*)

Точка входа для прерывания или продолжения обработки запроса. Эта функция должна возвращать объект *django.http.HttpResponse* или результат вызова родительского класса.

Начиная с релиза 5.3, было возможно написать этот метод асинхронным. Это должно использоваться в тех случаях, когда middleware делает запросы к базе данных или к кэшу. Однако такой компонент middleware должен быть исключен из bulk запросов.

[194] https://docs.djangoproject.com/en/3.2/topics/http/middleware/#writing-your-own-middleware

> **Предупреждение:** Никогда не делайте асинхронным middleware в цепях зависимостей. Они разработаны, чтобы отправлять независимые запросы к внешним ресурсам.
>
> Установите `async_capable` в `True` и `sync_capable` в `False` для таких middleware.

Параметры
request (*django.http.HttpRequest*[195]) – Объект HTTP-запроса, созданный из клиентского запроса.

Тип результата
django.http.HttpResponse[196]

handler(*request*, *response*)

Handler ответа. Метод для обработки ответов.

Параметры
- **request** (*django.http.HttpRequest*[197]) – Объект HTTP-запроса.
- **response** (*django.http.HttpResponse*[198]) – Объект HTTP-ответа, который будет отправлен клиенту.

Результат
Обработанный объект ответа.

Тип результата
django.http.HttpResponse[199]

request_handler(*request*)

Handler запроса. Вызывается перед обработкой запроса view.

Параметры
request (*django.http.HttpRequest*[200]) – Объект HTTP-запроса, созданный из клиентского запроса.

Результат
Обработанный объект запроса.

Тип результата
django.http.HttpRequest[201]

[195] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[196] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[197] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[198] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[199] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[200] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[201] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest

3.2.9 Filter Backend'ы

Filter Backend'ы[202] используются для изменения queryset'а модели. Чтобы создать пользовательский filter backend (т.е. аннотировать queryset модели), следует наследоваться от *vstutils.api.filter_backends.VSTFilterBackend* и переопределить `vstutils.api.filter_backends.VSTFilterBackend.filter_queryset()`. В некоторых случаях также стоит переопределить `vstutils.api.filter_backends.VSTFilterBackend.get_schema_fields()`.

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

Backend, фильтрующий queryset по колонке из свойства *deep_parent_field* модели. Значение для фильтрации должно быть передано в query-параметре *__deep_parent*.

Если параметр отсутствует, то никакие фильтры не применяются.

Если параметр - это пустое значение (*/?__deep_parent=*), то возвращаются объекты, не имеющие родителя (значение поля, чье имя хранится в свойстве модели *deep_parent_field*, равно None).

Этот filter backend и вложенное view автоматически добавляются в случае, если модель имеет свойство *deep_parent_field*.

Пример:

```
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'
```

В примере выше если мы добавим эту модель под путь „*deep*“, следующие view будут созданы: */deep/* и */deep/{id}/deepnested/*.

Filter backend, который может быть использован как */deep/?__deep_parent=1*, и будет возвращать все объекты *DeepNestedModel*, чьи родительские первичные ключи равны *1*.

Вы также можете использовать generic-view DRF. Для этого все еще нужно задать *deep_parent_field* вашей модели и вручную добавить *DeepViewFilterBackend* в список filter_backends[203].

class vstutils.api.filter_backends.**HideHiddenFilterBackend**

Filter Backend, убирающий из queryset все объекты, у которых задан атрибут hidden=True.

filter_queryset(*request*, *queryset*, *view*)

Очищает объекты со атрибутом hidden из queryset'а.

class vstutils.api.filter_backends.**SelectRelatedFilterBackend**

Filter Backend, автоматически вызывающий prefetch_related и select_related для всех отношений в queryset'е.

filter_queryset(*request*, *queryset*, *view*)

Выполняет select и prefetch в queryset'е.

[202] https://www.django-rest-framework.org/api-guide/filtering/#djangofilterbackend
[203] https://www.django-rest-framework.org/api-guide/filtering/#djangofilterbackend

class vstutils.api.filter_backends.**VSTFilterBackend**

Базовый класс filter backend'а, от которого следует наследоваться. Пример:

```
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()))
```

В данном примере Filter Backend аннотирует время в текущем часовом поясе в queryset'е используемой модели.

В некоторых случаях может быть необходимо передать параметр из query запроса. Чтобы определить этот параметр в схеме, вы должны перегрузить функцию get_schema_operation_parameters и указать список параметров, которые нужно использовать.

Пример:

```
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,
                    }
                },
            ]
```

В данном примере Filter Backend аннотирует время в текущем часовом поясе в queryset'е используемой модели именем поля из query *constant*.

get_schema_operation_parameters(*view*)

Вы также можете создать элементы управления фильтрами доступными для автогенерации схемы, предоставляемой REST-фреймворком, реализуя этот метод. Метод должен возвращать список OpenAPI сопоставлений схемы.

3.3 Celery

Celery - это распределенная очередь задач. Он используется для запуска задач асинхронно в отдельном worker'е. Чтобы узнать больше о Celery, смотрите официальную документацию[204]. Для работы функций vstutils, связанных с Celery, необходимо указать секции [rpc] and [worker] в settings.ini. Также вам понадобится установить дополнительные [rpc] зависимости.

3.3.1 Tasks

class vstutils.tasks.**TaskClass**

Обертка для класса BaseTask из Celery. Использование такое же, как и стандартного класса, однако вы можете запустить задачу без необходимости создавать экземпляр с помощью метода *TaskClass.do()*.

Пример:

```
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())
```

Теперь вы можете вызвать задачу несколькими методами:

- вызвав `Foo.do(*args, **kwargs)`
- получить зарегистрированный экземпляр задачи можно так: app.tasks[„full_path.to.task.class.Foo“]

Также вы можете сделать вашу зарегистрированную задачу периодической. Для этого нужно добавить ее CELERY_BEAT_SCHEDULE в 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*)

Метод, который посылает сигнал запуска удаленной задаче celery. Все аргументы будут переданы методу задачи *TaskClass.run()*.

Тип результата
`celery.result.AsyncResult`

property name

свойство для правильного выполнения Celery-задачи, нужно для работы метода *TaskClass.do()*

[204] https://docs.celeryproject.org/en/stable/

run (**args*, ***kwargs*)

Тело задачи выполняется worker'ами.

3.4 Endpoint

Endpoint-view имеет две цели: выполнение bulk-запросов и предоставление схемы OpenAPI.

URL endpoint'a - `/{API_URL}/endpoint/`, например значение с настройками по умолчанию - `/api/endpoint/`.

`API_URL` может быть изменен в `settings.py`.

class vstutils.api.endpoint.**EndpointViewSet** (***kwargs*)

Стандартный viewset API-endpoint'a.

get (*request*)

Возвращает ответ в виде Swagger UI или OpenAPI json-схему, если указан ?format=openapi

Параметры
request (vstutils.api.endpoint.BulkRequestType) –

Тип результата
django.http.response.HttpResponse

get_client (*request*)

Возвращает тестового клиента, гарантируя, что если bulk-запрос выполнен от аутентифицированного пользователя, то тестовый клиент будет аутентифицирован тем же самым пользователем.

Параметры
request (vstutils.api.endpoint.BulkRequestType) –

Тип результата
vstutils.api.endpoint.BulkClient

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

Возвращает экземпляр сериализатора, который должен быть использован для валидации и десериализации входных данных, и сериализации выходных данных.

Тип результата
vstutils.api.endpoint.OperationSerializer

get_serializer_context (*context*)

Дополнительный контекст, предоставляемый классу сериализатора.

Тип результата
dict[205]

operate (*operation_data*, *context*)

Метод, используемый для обработки одной операции и возвращающий ее результат

Параметры

- **operation_data** (typing.Dict[206]) –
- **context** (typing.Dict[207]) –

Тип результата
typing.Tuple[208][typing.Dict[209], typing.SupportsFloat[210]]

post (*request*)

Выполнить транзакционный bulk-запрос

Параметры
request (vstutils.api.endpoint.BulkRequestType) –

Тип результата
vstutils.api.responses.BaseResponseClass

put (*request*, *allow_fail=True*)

Выполнить нетранзакционный bulk-запрос

Параметры
request (vstutils.api.endpoint.BulkRequestType) –

Тип результата
vstutils.api.responses.BaseResponseClass

serializer_class

Класс сериализатора одной операции.

alias of `OperationSerializer`

versioning_class

alias of `QueryParameterVersioning`

3.4.1 Bulk-запросы

Bulk-запрос позволяет вам отсылать несколько запросов к api в одном. Он принимает json-список операций.

Метод	Транзакционный (все операции в одной транзакции)	Синхронный (операции выполняются одна за другой в указанном порядке)
`PUT /{API_URL}/endpoint/`	НЕТ	ДА
`POST /{API_URL}/endpoint/`	ДА	ДА
`PATCH /{API_URL}/endpoint/`	НЕТ	НЕТ

Параметры одной операции (обязательный параметр помечается *):

- `method`* - http-метод запроса
- `path`* - путь запроса, может быть типа `str` или `list`
- `data` - данные для отправки
- `query` - query-параметры типа `str`
- `let` - строка с именем переменной (используется для доступа к результату ответа в шаблонах)

[205] https://docs.python.org/3.6/library/stdtypes.html#dict
[206] https://docs.python.org/3.6/library/typing.html#typing.Dict
[207] https://docs.python.org/3.6/library/typing.html#typing.Dict
[208] https://docs.python.org/3.6/library/typing.html#typing.Tuple
[209] https://docs.python.org/3.6/library/typing.html#typing.Dict
[210] https://docs.python.org/3.6/library/typing.html#typing.SupportsFloat

- `headers` - словарь с заголовками, которые будут переданы в запрос (ключ - имя заголовка, значение - строка со значением заголовка.
- `version` - `str` с указанной версией api, если не задано, то используется `VST_API_VERSION`

Предупреждение: В предыдущих версиях имена заголовков должны были соответствовать спецификации CGI[211] (например, `CONTENT_TYPE`, `GATEWAY_INTERFACE`, `HTTP_*`).

Начиная с версии 5.3 и после миграции на Django 4 имена должны соответствовать HTTP спецификации вместо CGI.

В любой параметр запроса вы можете вставить результат значения предыдущей операции (`<<{OPERATION_NUMBER or LET_VALUE}[path][to][value]>>`), например:

```
[
    {"method": "post", "path": "user", "data": {"name": "User 1"}),
    {"method": "delete", "version": "v2", "path": ["user", "<<0[data][id]>>"]}
]
```

Результат bulk-запроса - это список json-объектов, описывающих операцию:

- `method` - http-метод
- `path` - путь запроса, всегда строка
- `data` - данные, которые нужно отправить
- `status` - код состояния ответа

Транзакционный bulk-запрос возвращает `502 BAG GATEWAY` и делает откат к состоянию до запроса после первого неудачного запроса.

Предупреждение: Если вы отправили нетранзакционный bulk-запрос, вы получите код 200 и должны будете проверить статус каждого ответа операции отдельно.

3.4.2 Схема OpenAPI

Запрос на `GET /{API_URL}/endpoint/` возвращает Swagger UI.

Запрос на `GET /{API_URL}/endpoint/?format=openapi` возвращает схему OpenAPI в формате json. Также вы можете указать нужную версию схемы, используя query-параметр `version`

Для изменения схемы после ее генерации и перед отправкой пользователю используйте хуки. Напишите одну или несколько функций, каждая из которых принимает 2 именованных аргумента:

- `request` - объект запроса пользователя.
- `schema` - ordered dict, содержащий схему OpenAPI.

Примечание: Иногда хуки могут выбросить исключение; чтобы сохранить цепочку модификации данных, такие исключения обрабатываются. Изменения, сделанные в схеме перед выбросом исключения, в любом случае сохраняются.

Пример хука:

[211] https://www.w3.org/CGI/

```
def hook_add_username_to_guiname(request, schema):
    schema['info']['title'] = f"{request.username} - {schema['info']['title']}"
```

Чтобы присоединить хук(-и) к вашему приложению, добавьте строку импорта вашей функции в список `OPENAPI_HOOKS` в `settings.py`

```
OPENAPI_HOOKS = [
    '{{appName}}.openapi.hook_add_username_to_guiname',
]
```

3.5 Фреймворк для тестирования

Фреймворк VST Utils включает в себя хелпер в базовом тест-кейс классе и улучшает поддержку механизма отправки запросов. На практике это означает, что для отправления bulk-запроса на endpoint нет необходимости создавать и инициализировать test client, а можно сразу делать запрос.

```
endpoint_results = self.bulk([
    # list of endpoint requests
])
```

3.5.1 Создание тест-кейса

Модуль `test.py` содержит классы тест-кейсов, основанные на *vstutils.tests.BaseTestCase*. На текущий момент мы официально поддерживаем два подхода к написанию тестов: классический и с помощью оберток запросов с проверкой выполнения и runtime-оптимизацией bulk-запросов с ручной проверкой значений.

3.5.2 Простой пример с классическими тестами

Например, если у вас endpoint вида `/api/v1/project/` и модель Project, вы можете написать такой тест:

```
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
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        )
        # 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))
```

Этот пример демонстрирует функциональность стандартного тест-кейс класса. Проекты по умолчанию инициализируются для получения наиболее быстрого и эффективного результата. Рекомендуется разбивать тесты на разные сущности в разные классы. В данном примере показан классический подход к тестированию, однако вы можете использовать bulk-запросы в ваших тестах.

3.5.3 Bulk-запросы в тестах

Система bulk-запросов хорошо подходит для тестирования и запуска валидных запросов. Предыдущий пример может быть переписан так:

```
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)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        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))
```

В этом случае хотя мы и получили больше кода, однако тесты стали ближе к процессу использования приложения в графическом интерфейсе, потому что проекты vstutils используют `/api/endpoint/` для выполнения запросов. Так или иначе, bulk-запросы выполняются заметно быстрее благодаря оптимизации, которую они выполняют под капотом. Время выполнения теста, в котором используется bulk меньше по сравнению с тестом, использующим стандартный механизм.

3.5.4 API тест-кейса

class vstutils.tests.**BaseTestCase**(*methodName='runTest'*)

Основной тест-кейс класс расширяет `django.test.TestCase`[212].

assertCheckDict(*first*, *second*, *msg=None*)

Падает, если два поля в словаре не равны по определению оператора „==“. Проверяет первое поле на пустоту и на равенство со вторым полем

Параметры

- **first** (`typing.Dict`[213]) –
- **second** (`typing.Dict`[214]) –
- **msg** (`str`[215]) –

assertCount(*iterable*, *count*, *msg=None*)

Вызывает `len()`[216] через `iterable` и проверяет равенство с `count`.

Параметры

- **iterable** (`typing.Sized`[217]) – любой итерируемый объект, который может быть отправлен в `len()`[218].
- **count** (`int`[219]) – ожидаемый результат.
- **msg** (`typing.Any`[220]) – сообщение об ошибке

assertRCode(*resp*, *code=200*, **additional_info*)

Падает, если коды ответа не совпадают. Сообщением является тело ответа.

Параметры

- **resp** (*`django.http.HttpResponse`*[221]) – объект ответа
- **code** (*`int`*[222]) – ожидаемый код

bulk(*data*, *code=200*, ***kwargs*)

Делает нетранзакционный bulk-запрос и проверяет код состояния (200 по умолчанию)

Параметры

- **data** (`typing.Union`[223][`typing.List`[224][`typing.Dict`[225][`str`[226], `typing.Any`[227]]], `str`[228], `bytes`[229], `bytearray`[230]) – данные запроса

- **code** (int[231]) – http-статус для проверки
- **kwargs** – именованные аргументы для *get_result()*

Тип результата
typing.Union[232][typing.List[233][typing.Dict[234][str[235], typing.Any[236]]], str[237], bytes[238], bytearray[239], typing.Dict[240], typing.Sequence[241][typing.Union[242][typing.List[243][typing.Dict[244][str[245], typing.Any[246]]], str[247], bytes[248], bytearray[249]]]

Результат
bulk-ответ

bulk_transactional(*data*, *code=200*, ***kwargs*)

Делает транзакционный bulk-запрос и проверяет код состояния (200 по умолчанию)

Параметры

- **data** (typing.Union[250][typing.List[251][typing.Dict[252][str[253], typing.Any[254]]], str[255], bytes[256], bytearray[257]) – данные запроса
- **code** (int[258]) – http-статус для проверки
- **kwargs** – именованные аргументы для *get_result()*

Тип результата
typing.Union[259][typing.List[260][typing.Dict[261][str[262], typing.Any[263]]], str[264], bytes[265], bytearray[266], typing.Dict[267], typing.Sequence[268][typing.Union[269][typing.List[270][typing.Dict[271][str[272], typing.Any[273]]], str[274], bytes[275], bytearray[276]]]

Результат
bulk-ответ

call_registration(*data*, ***kwargs*)

Функция для вызова регистрации. Просто передайте данные формы вместе с заголовками.

Параметры

- **data** (*dict*[277]) – Данные регистрации с формы.
- **kwargs** – именованные аргументы вместе с заголовками запроса.

details_test(*url*, ***kwargs*)

Тест на получение детальной записи модели. При задании дополнительных именованных аргументов метод проверит их на равенство с полученными данными. Использует метод *get_result()*.

Параметры

- **url** – url детальной записи. Например: `/api/v1/project/1/` (где `1` - это уникальный идентификатор проекта). Вы можете использовать *get_url()* для построения url.
- **kwargs** – параметры для проверки (ключ - имя поля, значение - значение поля).

endpoint_call(*data=None*, *method='get'*, *code=200*, ***kwargs*)

Делает запрос на endpoint и проверяет код состояния ответа, если он задан (200 по умолчанию). Использует *get_result()*.

Параметры

- **data** (typing.Union[278][typing.List[279][typing.Dict[280][str[281], typing.Any[282]]], str[283], bytes[284], bytearray[285]) – данные запроса

- **method** (str[286]) – метод http-запроса
- **code** (int[287]) – http-статус для проверки
- **query** – словарь с данными query (работает только с *get*)

Тип результата
typing.Union[288][typing.List[289][typing.Dict[290][str[291], typing.Any[292]]], str[293], bytes[294], bytearray[295], typing.Dict[296], typing.Sequence[297][typing.Union[298][typing.List[299][typing.Dict[300][str[301], typing.Any[302]]], str[303], bytes[304], bytearray[305]]]

Результат
bulk-ответ

endpoint_schema (***kwargs*)

Делает запрос на схему. Возвращает словарь с данными swagger.

Параметры
version – Версия API для парсера схемы.

get_count (*model*, ***kwargs*)

Простая обертка над *get_model_filter()*, возвращающая счетчик объектов.

Параметры

- **model** (*str*[306], *django.db.models.Model*[307]) – строка, содержащая имя модели (если атрибут `model` установлен в класс тест-кейса), импорт модуля, `app.ModelName` или `django.db.models.Model`[308].
- **kwargs** – именованные аргументы для `django.db.models.query.QuerySet.filter()`[309].

Результат
количество объектов в базе данных.

Тип результата
int[310]

get_model_class (*model*)

Получение класса модели по строке или получение аргумента модели.

Параметры
model (*str*[311], *django.db.models.Model*[312]) – строка, содержащая имя модели (если атрибут `model` установлен в класс тест-кейса), импорт модуля, `app.ModelName` или `django.db.models.Model`[313].

Результат
Класс модели.

Тип результата
django.db.models.Model[314]

get_model_filter (*model*, ***kwargs*)

Простая обертка над *get_model_class()*, возвращающая фильтрованный queryset из модели.

Параметры

- **model** (*str*[315], *django.db.models.Model*[316]) – строка, содержащая имя модели (если атрибут `model` установлен в класс тест-кейса), импорт модуля, `app.ModelName` или `django.db.models.Model`[317].

- **kwargs** – именованные аргументы для `django.db.models.query.QuerySet.filter()`[318].

Тип результата
django.db.models.query.QuerySet[319]

get_result(*rtype*, *url*, *code=None*, **args*, ***kwargs*)

Запускает и проверяет код ответа для запроса, возвращает распарсенный результат запроса. Данный метод действует следующим образом:

- Тестирует авторизацию клиента (вместе с `user`, который создается в `setUp()`).
- Выполняет запрос (отправляет аргументы и именованные аргументы в метод запроса).
- Парсит результат (конвертирует строку json в объект python).
- Проверяет http-код состояния с помощью *`assertRCode()`* (если вы его не указали, будет выбран соответствующий код для выполняемого метода из стандартного набора *`std_codes`*).
- Деавторизация пользователя.
- Возвращение распарсенного результата.

Параметры

- **rtype** – тип запроса (методы из Client cls): get, post и т.д.
- **url** – запрошенный url в виде строки или кортежа для *`get_url()`*. Вы можете использовать *`get_url()`* для построения url или задать его полной строкой.
- **code** (`int`[320]) – ожидаемый код возврата из запроса.
- **relogin** – выполнение авторизации и деавторизации перед каждым вызовом. По умолчанию `True`.
- **args** – дополнительные аргументы для метода запроса класса Client.
- **kwargs** – дополнительные именованные аргументы для метода запроса класса Client.

Тип результата
`typing.Union`[321][`typing.List`[322][`typing.Dict`[323][`str`[324], `typing.Any`[325]]], `str`[326], `bytes`[327], `bytearray`[328], `typing.Dict`[329], `typing.Sequence`[330][`typing.Union`[331][`typing.List`[332][`typing.Dict`[333][`str`[334], `typing.Any`[335]]], `str`[336], `bytes`[337], `bytearray`[338]]]]

Результат
результат запроса.

get_url(**items*)

Функция для создания пути url, основанного на настройках `VST_API_URL` и `VST_API_VERSION`. Без аргументов возвращает путь к версии api по умолчанию.

Тип результата
`str`[339]

Результат
строка вида `/api/v1/.../.../` где `...` - аргументы функции.

list_test(*url*, *count*)

Тест на получение списка моделей. Проверяет только количество записей. Использует метод *`get_result()`*.

Параметры

- **url** – url абстрактного слоя. Например: `/api/v1/project/`. Вы можете использовать *`get_url()`* для построения url.
- **count** – количество объектов в базе данных.

models = None

Атрибут с модулем моделей проекта по умолчанию.

classmethod patch(**args*, ***kwargs*)

Простая обертка над `unittest.mock.patch()`[340].

Тип результата
`typing.ContextManager`[341][`unittest.mock.Mock`[342]]

classmethod patch_field_default(*model*, *field_name*, *value*)

Этот метод помогает найти значение по умолчанию в поле модели. Он очень полезен для полезен для полей DateTime, где по умолчанию установлено `django.utils.timezone.now()`[343].

Параметры

- **model** (`django.db.models.base.Model`) –
- **field_name** (`str`[344]) –
- **value** (`typing.Any`[345]) –

Тип результата
`typing.ContextManager`[346][`unittest.mock.Mock`[347]]

random_name()

Простая функция, возвращающая строку uuid1.

Тип результата
`str`[348]

std_codes: typing.Dict[349]**[str**[350]**, int**[351]**] = {'delete': 204, 'get': 200, 'patch': 200, 'post': 201}**

Стандартный http-код для различных http-методов. Использует *`get_result()`*

class user_as(*testcase*, *user*)

Контекст для выполнения bulk или чего-либо еще от некоторого пользователя. Контекстный менеджер переопределяет `self.user` в TestCase'е и возвращает изменения после выхода из него.

Параметры
user (*`django.contrib.auth.models.AbstractUser`*[352]) – новый объект пользователя, от которого будет выполнение.

[212] https://docs.djangoproject.com/en/4.2/topics/testing/tools/#django.test.TestCase
[213] https://docs.python.org/3.6/library/typing.html#typing.Dict
[214] https://docs.python.org/3.6/library/typing.html#typing.Dict
[215] https://docs.python.org/3.6/library/stdtypes.html#str
[216] https://docs.python.org/3.6/library/functions.html#len
[217] https://docs.python.org/3.6/library/typing.html#typing.Sized
[218] https://docs.python.org/3.6/library/functions.html#len
[219] https://docs.python.org/3.6/library/functions.html#int
[220] https://docs.python.org/3.6/library/typing.html#typing.Any
[221] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[222] https://docs.python.org/3.6/library/functions.html#int
[223] https://docs.python.org/3.6/library/typing.html#typing.Union
[224] https://docs.python.org/3.6/library/typing.html#typing.List
[225] https://docs.python.org/3.6/library/typing.html#typing.Dict
[226] https://docs.python.org/3.6/library/stdtypes.html#str
[227] https://docs.python.org/3.6/library/typing.html#typing.Any
[228] https://docs.python.org/3.6/library/stdtypes.html#str
[229] https://docs.python.org/3.6/library/stdtypes.html#bytes
[230] https://docs.python.org/3.6/library/stdtypes.html#bytearray
[231] https://docs.python.org/3.6/library/functions.html#int
[232] https://docs.python.org/3.6/library/typing.html#typing.Union
[233] https://docs.python.org/3.6/library/typing.html#typing.List
[234] https://docs.python.org/3.6/library/typing.html#typing.Dict
[235] https://docs.python.org/3.6/library/stdtypes.html#str
[236] https://docs.python.org/3.6/library/typing.html#typing.Any
[237] https://docs.python.org/3.6/library/stdtypes.html#str
[238] https://docs.python.org/3.6/library/stdtypes.html#bytes
[239] https://docs.python.org/3.6/library/stdtypes.html#bytearray
[240] https://docs.python.org/3.6/library/typing.html#typing.Dict
[241] https://docs.python.org/3.6/library/typing.html#typing.Sequence
[242] https://docs.python.org/3.6/library/typing.html#typing.Union
[243] https://docs.python.org/3.6/library/typing.html#typing.List
[244] https://docs.python.org/3.6/library/typing.html#typing.Dict
[245] https://docs.python.org/3.6/library/stdtypes.html#str
[246] https://docs.python.org/3.6/library/typing.html#typing.Any
[247] https://docs.python.org/3.6/library/stdtypes.html#str
[248] https://docs.python.org/3.6/library/stdtypes.html#bytes
[249] https://docs.python.org/3.6/library/stdtypes.html#bytearray
[250] https://docs.python.org/3.6/library/typing.html#typing.Union
[251] https://docs.python.org/3.6/library/typing.html#typing.List
[252] https://docs.python.org/3.6/library/typing.html#typing.Dict
[253] https://docs.python.org/3.6/library/stdtypes.html#str
[254] https://docs.python.org/3.6/library/typing.html#typing.Any
[255] https://docs.python.org/3.6/library/stdtypes.html#str
[256] https://docs.python.org/3.6/library/stdtypes.html#bytes
[257] https://docs.python.org/3.6/library/stdtypes.html#bytearray
[258] https://docs.python.org/3.6/library/functions.html#int
[259] https://docs.python.org/3.6/library/typing.html#typing.Union
[260] https://docs.python.org/3.6/library/typing.html#typing.List
[261] https://docs.python.org/3.6/library/typing.html#typing.Dict
[262] https://docs.python.org/3.6/library/stdtypes.html#str
[263] https://docs.python.org/3.6/library/typing.html#typing.Any
[264] https://docs.python.org/3.6/library/stdtypes.html#str
[265] https://docs.python.org/3.6/library/stdtypes.html#bytes
[266] https://docs.python.org/3.6/library/stdtypes.html#bytearray
[267] https://docs.python.org/3.6/library/typing.html#typing.Dict
[268] https://docs.python.org/3.6/library/typing.html#typing.Sequence
[269] https://docs.python.org/3.6/library/typing.html#typing.Union
[270] https://docs.python.org/3.6/library/typing.html#typing.List
[271] https://docs.python.org/3.6/library/typing.html#typing.Dict
[272] https://docs.python.org/3.6/library/stdtypes.html#str
[273] https://docs.python.org/3.6/library/typing.html#typing.Any
[274] https://docs.python.org/3.6/library/stdtypes.html#str
[275] https://docs.python.org/3.6/library/stdtypes.html#bytes
[276] https://docs.python.org/3.6/library/stdtypes.html#bytearray
[277] https://docs.python.org/3.6/library/stdtypes.html#dict
[278] https://docs.python.org/3.6/library/typing.html#typing.Union
[279] https://docs.python.org/3.6/library/typing.html#typing.List
[280] https://docs.python.org/3.6/library/typing.html#typing.Dict
[281] https://docs.python.org/3.6/library/stdtypes.html#str
[282] https://docs.python.org/3.6/library/typing.html#typing.Any
[283] https://docs.python.org/3.6/library/stdtypes.html#str
[284] https://docs.python.org/3.6/library/stdtypes.html#bytes
[285] https://docs.python.org/3.6/library/stdtypes.html#bytearray

3.6 Утилиты

Здесь представлен проверенный набор утилит для разработки. Они включают в себя код, который так или иначе будет полезен по мере разработки. Vstutils использует большинство из этих функций под капотом.

class vstutils.utils.**BaseEnum**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

BaseEnum расширяет класс *Enum* и используется для создания enum-подобных объектов, которые могут использоваться django-сериализаторами или django-моделями.

Пример:

```
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)
```

Примечание: вы можете установить значение как `BaseEnum.LOWER` или ``BaseEnum.UPPER``. Однако в обычных случаях рекомендуется использовать `BaseEnum.SAME` для оптимизации памяти.

---

class vstutils.utils.**BaseVstObject**

Стандартная миксина для пользовательских объектов, которым нужны настройки или кэш.

classmethod get_django_settings(*name*, *default=None*)

Получить параметры из настроек Django.

Параметры

- **name** (*str*[353]) – название параметра
- **default** (*object*[354]) – значение параметра по умолчанию

Результат

Параметр из настроек Django.

class vstutils.utils.**Dict**

Обертка над *dict*, возвращающая JSON при преобразовании в строку.

class vstutils.utils.**Executor**(*stdout=-1*, *stderr=-2*, ***environ_variables*)

Исполнитель команд с выводом и обработкой строк в реальном времени. По умолчанию и замыслу исполнитель инициализирует строковый атрибут `output`, который будет изменен оператором += с новыми

[353] https://docs.python.org/3.6/library/stdtypes.html#str
[354] https://docs.python.org/3.6/library/functions.html#object

строками с помощью метода *Executor.write_output()*. Переопределите метод, если нужно изменить поведение.

Класс исполнителя поддерживает периодеческий (0.01 сек) процесс обработки и выполняет некоторые проверки путем переопределения метода *Executor.working_handler()*. Если вы хотите отключить это поведение, переопределите метод значением None или используйте *UnhandledExecutor*.

Параметры
environ_variables (str[355]) –

exception CalledProcessError(*returncode*, *cmd*, *output=None*, *stderr=None*)

Выбрасывается, когда run() вызывается вместе с check=True и процесс возвращает код возврата отличный от нуля.

Атрибуты:
cmd, returncode, stdout, stderr, output

property stdout

Псевдоним для выходного атрибута, чтобы соответствовать stderr

async aexecute(*cmd*, *cwd*, *env=None*)

Выполняет команды и выводит их результат. Асинхронная реализация.

Параметры

- **cmd** – – список cmd-команд и аргументов
- **cwd** – – рабочая директория
- **env** – – дополнительные переменные окружения, которые перезаписывают переменные по умолчанию

Результат
– строка, содержащая полный вывод

execute(*cmd*, *cwd*, *env=None*)

Выполняет команды и выводит их результат.

Параметры

- **cmd** – – список cmd-команд и аргументов
- **cwd** – – рабочая директория
- **env** – – дополнительные переменные окружения, которые перезаписывают переменные по умолчанию

Результат
– строка, содержащая полный вывод

async post_execute(*cmd*, *cwd*, *env*, *return_code*)

Запускается после завершения выполнения.

Параметры

- **cmd** – – список cmd-команд и аргументов
- **cwd** – – рабочая директория
- **env** – – дополнительные переменные окружения, которые перезаписывают переменные по умолчанию
- **return_code** – – код возврата выполненного процесса

async pre_execute(*cmd*, *cwd*, *env*)

Запускатеся перед началом выполнения.

Параметры

- **cmd** – – список cmd-команд и аргументов
- **cwd** – – рабочая директория
- **env** – – дополнительные переменные окружения, которые перезаписывают переменные по умолчанию

async working_handler(*proc*)

Дополнительный обработчик для запусков.

Параметры
proc (*asyncio.subprocess.Process*) – запущенный процесс

write_output(*line*)

Параметры
line (*str*[356]) – – строка вывода команды

Результат
None

Тип результата
None

class vstutils.utils.**KVExchanger**(*key*, *timeout=None*)

Класс для передачи данных с использованием быстрого (кэш-подобного) хранилища между сервисами. Использует тот же самый кэш-бэкенд, что и Lock.

class vstutils.utils.**Lock**(*id*, *payload=1*, *repeat=1*, *err_msg=''*, *timeout=None*)

Класс Lock предназначен для работы с несколькими задачами. Основан на *KVExchanger*. Lock позволяет только одному потоку войти в заблокированную и совместно используемую часть между приложениями, использующими один кэш блокировок (см. также [locks]).

Параметры

- **id** (*int*[357], *str*[358]) – – уникальный id блокировки.
- **payload** – – дополнительная информация о блокировке. Должна быть значением, равным True при приведении к булевому типу.
- **repeat** (*int*[359]) – – время ожидания lock.release. По умолчанию 1 секунда.
- **err_msg** (*str*[360]) – – сообщение для ошибки AcquireLockException.

Примечание:

- Использует django.core.cache и настройки в *settings.py*
- Имеет Lock.SCHEDULER и Lock.GLOBAL id

Пример:

[355] https://docs.python.org/3.6/library/stdtypes.html#str
[356] https://docs.python.org/3.6/library/stdtypes.html#str

```
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`
    # and ``err_msg`` value as text.
    some_code_execution()
    # ``lock`` object will has been automatically released after
    # exiting from context.
```

Другой пример без использования контекстного менеджера:

```
from vstutils.utils import Lock

# locked block after locked object created
lock = Lock("some_lock_identifier", repeat=30, err_msg="Locked by another␣
↪proccess")
# deleting of object calls ``lock.release()`` which release and remove lock␣
↪from id.
del lock
```

exception AcquireLockException

Исключение, которое будет выброшено в случае неосвобождения блокировки.

class vstutils.utils.**ModelHandlers**(*type_name*, *err_message=None*)

Обработчики для некоторых моделей, таких как „INTEGRATIONS“ или „REPO_BACKENDS“. Основан на *ObjectHandlers*, но больше сосредоточен на работе с моделями. Все handler-бэкенды получают объект модели по первому аргументу.

Атрибуты:

Параметры

- **objects** (*dict*[361]) – – словарь объектов, например {<name>: <backend_class>}
- **keys** (*list*[362]) – – имена поддерживаемых бэкендов
- **values** (*list*[363]) – – поддерживаемые классы бэкендов
- **type_name** – Имя для бэкенда, наподобие ключа в словаре.

get_object(*name*, *obj*)

Параметры

- **name** – – строковое имя бэкенда
- **name** – str
- **obj** (*django.db.models.Model*[364]) – – объект модели

[357] https://docs.python.org/3.6/library/functions.html#int
[358] https://docs.python.org/3.6/library/stdtypes.html#str
[359] https://docs.python.org/3.6/library/functions.html#int
[360] https://docs.python.org/3.6/library/stdtypes.html#str

Результат
объект бэкенда

Тип результата
object[365]

class vstutils.utils.**ObjectHandlers**(*type_name*, *err_message=None*)

Обертка обработчиков для получения объектов из некоторой структуры настроек.

Пример:

```
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')
```

Параметры
type_name (*str*[366]) – Имя для бэкенда, наподобие ключа в словаре.

backend(*name*)

Получить класс бэкенда

Параметры
name (*str*[367]) – – имя типа бэкенда

Результат
класс бэкенда

Тип результата
type[368],types.ModuleType[369],object[370]

[361] https://docs.python.org/3.6/library/stdtypes.html#dict
[362] https://docs.python.org/3.6/library/stdtypes.html#list
[363] https://docs.python.org/3.6/library/stdtypes.html#list
[364] https://docs.djangoproject.com/en/4.2/ref/models/instances/#django.db.models.Model
[365] https://docs.python.org/3.6/library/functions.html#object

class vstutils.utils.**Paginator**(*qs*, *chunk_size=None*)

Класс для разбиения запроса на небольшие запросы.

class vstutils.utils.**SecurePickling**(*secure_key=None*)

Защищенная pickle-обертка с использованием шифра Виженера.

> **Предупреждение:** В любом случае не используйте его с ненадежным средством передачи.

Пример:

```
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*)

Обработчик объекта для views в графическом интерфейсе. Использует *GUI_VIEWS* из settings.py. Основан на *ObjectHandlers*, но больше сосредоточен на urlpatterns.

Пример:

```
from vstutils.utils import URLHandlers


# By default gets from `GUI_VIEWS` in `settings.py`
urlpatterns = list(URLHandlers())
```

Параметры

type_name – Имя для бэкенда, наподобие ключа в словаре.

get_object(*name*, **argv*, ***kwargs*)

Получить объект кортежа url'ов для urls.py

Параметры

- **name** (*str*[371]) – регулярное выражение url'а
- **argv** – переопределенные аргументы
- **kwargs** – переопределенные kwarg'и

[366] https://docs.python.org/3.6/library/stdtypes.html#str
[367] https://docs.python.org/3.6/library/stdtypes.html#str
[368] https://docs.python.org/3.6/library/functions.html#type
[369] https://docs.python.org/3.6/library/types.html#types.ModuleType
[370] https://docs.python.org/3.6/library/functions.html#object

Результат
объект url'a

Тип результата
django.urls.re_path

class vstutils.utils.**UnhandledExecutor**(*stdout=-1*, *stderr=-2*, ***environ_variables*)

Класс, основанный на *Executor*, но с выключенным *working_handler*.

Параметры
environ_variables (str[372]) –

class vstutils.utils.**apply_decorators**(**decorators*)

Декоратор, оборачивающий метод или класс в список декораторов.

Пример:

```
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*)

Декоратор, который из метода класса делает классовый property.

Пример:

```
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
```

Параметры

- **fget** – Функция для получения значения атрибута.

[371] https://docs.python.org/3.6/library/stdtypes.html#str
[372] https://docs.python.org/3.6/library/stdtypes.html#str

- **fset** – Функция для установки значения атрибута.

vstutils.utils.**create_view**(*model*, ***meta_options*)

Простая функция для получения сгенерированного view стандартными средствами, но с перегруженными мета-параметрами. Этот метод позволяет полностью отказаться от создания прокси-моделей.

Пример:

```
from vstutils.utils import create_view

from .models import Host

# 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'
)
```

Примечание: Данный метод также рекомендуется применять в случаях, когда имеются проблемы с рекурсивными импортами.

Параметры

model (*Type[vstutils.models.BaseModel]*) – Класс модели с методом *.get_view_class*. Этот метод также имеет *vstutils.models.BModel*.

Тип результата

vstutils.api.base.GenericViewSet

vstutils.utils.**decode**(*key*, *enc*)

Декодировать строку из закодированной шифром Виженера.

Параметры

- **key** (*str*[373]) – – секретный ключ для кодирования
- **enc** (*str*[374]) – – закодированная строка для декодирования

Результат

– декодированная строка

Тип результата

str[375]

vstutils.utils.**deprecated**(*func*)

Данный декоратор может быть использован, чтобы пометить функцию как устаревшую. После этого ее вызов приведет к выдаче соответствующего предупреждения.

Параметры

func – любой вызываемый объект, который будет обернут и выдаст предупреждение об устаревании при вызове.

[373] https://docs.python.org/3.6/library/stdtypes.html#str
[374] https://docs.python.org/3.6/library/stdtypes.html#str
[375] https://docs.python.org/3.6/library/stdtypes.html#str

vstutils.utils.**encode**(*key*, *clear*)

Закодировать строку шифром Виженера.

Параметры

- **key** (*str*[376]) – – секретный ключ для кодирования
- **clear** (*str*[377]) – – чистое значение для кодирования

Результат
– закодированная строка

Тип результата
str[378]

vstutils.utils.**get_render**(*name*, *data*, *trans='en'*)

Рендеринг строки из шаблона.

Параметры

- **name** (*str*[379]) – – полное название шаблона
- **data** (*dict*[380]) – – словарь переменных для рендеринга
- **trans** (*str*[381]) – – перевод для рендера. По умолчанию „en“.

Результат
– отрендеренная строка

Тип результата
str[382]

vstutils.utils.**lazy_translate**(*text*)

Функция `lazy_translate` имеет то же поведение, что и *translate()*, но оборачивает его в lazy promise.

Это полезно, например, для перевода сообщений об ошибках в атрибутах класса, когда целевой язык еще неизвестен.

Параметры
text – Текстовое сообщение, которое должно быть переведено.

vstutils.utils.**list_to_choices**(*items_list*, *response_type=<class 'list'>*)

Метод, предназначенный для создания django-модели choices из плоского списка значений.

Параметры

- **items_list** – плоский список значений.
- **response_type** – тип приведения возвращаемого сопоставления

Результат
список кортежей из значений *items_list*

[376] https://docs.python.org/3.6/library/stdtypes.html#str
[377] https://docs.python.org/3.6/library/stdtypes.html#str
[378] https://docs.python.org/3.6/library/stdtypes.html#str
[379] https://docs.python.org/3.6/library/stdtypes.html#str
[380] https://docs.python.org/3.6/library/stdtypes.html#dict
[381] https://docs.python.org/3.6/library/stdtypes.html#str
[382] https://docs.python.org/3.6/library/stdtypes.html#str

class vstutils.utils.**model_lock_decorator**(***kwargs*)

Декоратор для функций, где kwarg „pk“ существует для блокировки по id.

Предупреждение:

- В случае ошибки блокировки выбрасывает `Lock.AcquireLockException`
- Метод должен иметь и быть вызван вместе с именованным аргументом `pk`.

class vstutils.utils.**raise_context**(**args*, ***kwargs*)

Контекст для игнорирования исключений.

class vstutils.utils.**raise_context_decorator_with_default**(**args*, ***kwargs*)

Контекст для предотвращения исключений и возврата значения по умолчанию.

Пример:

```
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*)

Контекст для перенаправления любого вывода в свой поток.

Примечание:

- В контексте возвращает объект потока.
- При выходе возвращает старые потоки.

---

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*)

Обертка над `django.core.mail.send_mail()`[383], предоставляющая дополнительные именованные аргументы.

vstutils.utils.**send_template_email**(*sync=False*, ***kwargs*)

Функция, выполняющая синхронную или асинхронную отправку электронной почты в зависимости от аргумента *sync* и переменной настроек «RPC_ENABLED». Вы можете использовать эту функцию для отправки сообщений, она отправляет сообщение асинхронно или синхронно. Если вы не установили настройки для Celery или не установили Celery, она отправляет письмо синхронно. Если установлен и настроен Celery, и аргумент *sync* функции установлен на *False*, она отправляет электронное письмо асинхронно.

[383] https://docs.djangoproject.com/en/4.2/topics/email/#django.core.mail.send_mail

Параметры

- **sync** – аргумент для определения, как отправлять электронную почту, асинхронно или синхронно.
- **subject** – тема письма.
- **email** – список строк или отдельная строка с адресами электронной почты получателей.
- **template_name** – относительный путь к шаблону в директории *templates*, должен включать расширение имени файла.
- **context_data** – словарь с контекстом для отображения шаблона сообщения.

vstutils.utils.**send_template_email_handler**(*subject*, *email_from*, *email*, *template_name*, *context_data=None*, ***kwargs*)

Функция для отправки электронной почты. Функция преобразует получателя в список и устанавливает контекст перед отправкой, если это возможно.

Параметры

- **subject** – тема письма.
- **email_from** – адрес отправителя, который будет указан в письме.
- **email** – список строк или отдельная строка с адресами электронной почты получателей.
- **template_name** – относительный путь к шаблону в директории *templates*, должен включать расширение имени файла.
- **context_data** – словарь с контекстом для отображения шаблона сообщения.
- **kwargs** – дополнительные именованные аргументы для *send_mail*.

Результат

Количество отправленных электронных писем.

class vstutils.utils.**tmp_file**(*data=''*, *mode='w'*, *bufsize=-1*, ***kwargs*)

Временный файл с сгенерированным и автоматически именем и удаленный по закрытии

Атрибуты:

Параметры

- **data** (*str*[384]) – – строка для записи во временный файл.
- **mode** (*str*[385]) – – режим открытия файла. По умолчанию *w*.
- **bufsize** (*int*[386]) – – размер буфера для tempfile.NamedTemporaryFile.
- **kwargs** – – другие именованные аргументы для tempfile.NamedTemporaryFile.

write(*wr_string*)

Записать в файл и очистить буфер

Параметры

wr_string (*str*[387]) – – записываемая строка

Результат

None

Тип результата

None

class vstutils.utils.**tmp_file_context**(**args*, ***kwargs*)

Объект контекста для работы с tmp_file. Автоматическое закрывается при выходе из контекста и удаляется файл, если он все еще существует.

Данный менеджер контекста работает с class:.*tmp_file*

vstutils.utils.**translate**(*text*)

Функция `translate` поддерживает динамический перевод сообщения с использованием стандартных механизмов i18n в vstutils.

Использует функцию `django.utils.translation.get_language()`[388] для получения кода языка и пытается получить перевод из списка доступных.

Параметры

text – Текстовое сообщение, которое должно быть переведено.

[384] https://docs.python.org/3.6/library/stdtypes.html#str
[385] https://docs.python.org/3.6/library/stdtypes.html#str
[386] https://docs.python.org/3.6/library/functions.html#int
[387] https://docs.python.org/3.6/library/stdtypes.html#str
[388] https://docs.djangoproject.com/en/4.2/ref/utils/#django.utils.translation.get_language

4

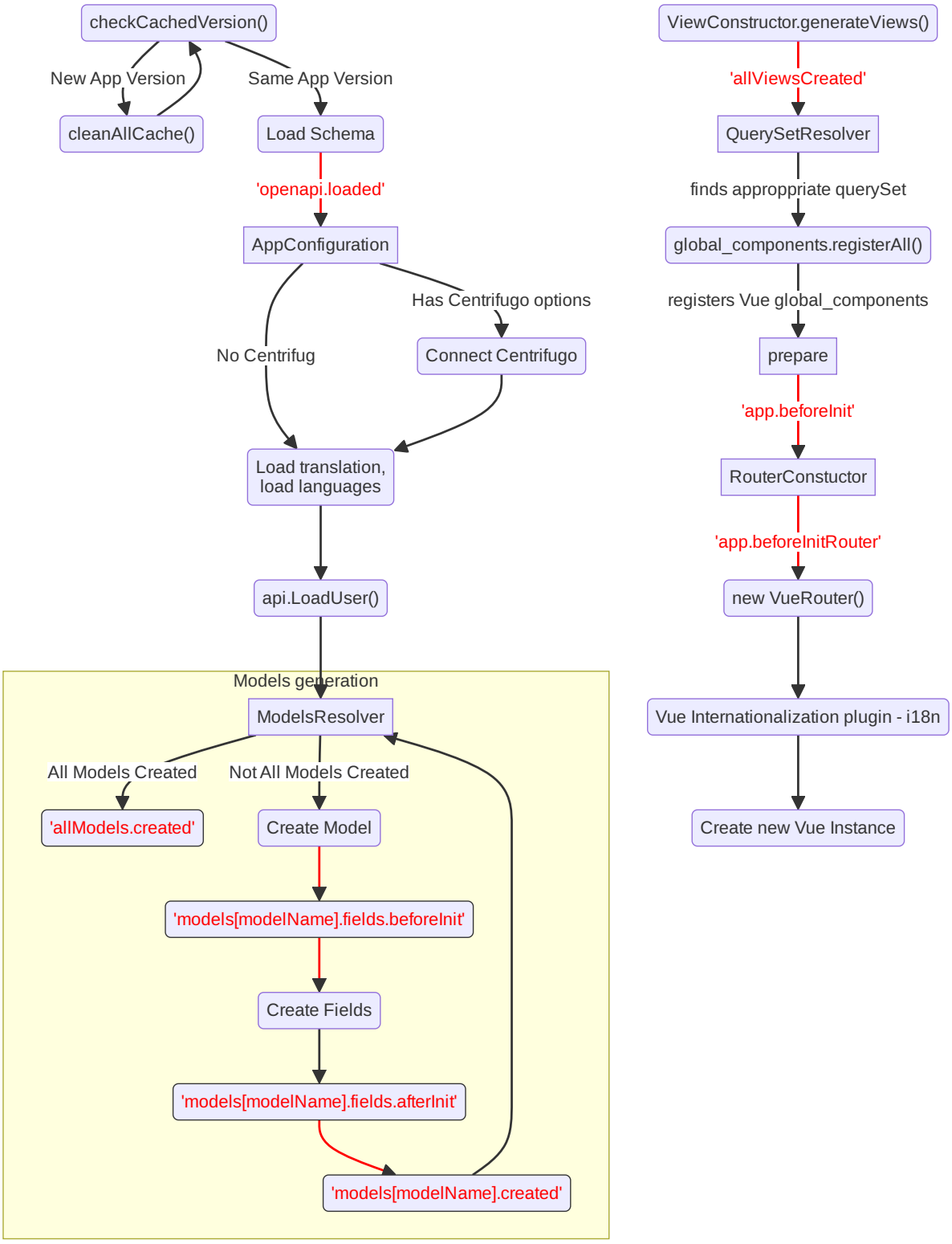
Frontend Quickstart

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 «*Настройки клиента Centrifugo*»;
    - 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)

(продолжение с предыдущей страницы)

```
    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.

Примечание: 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[389].

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:

[389] 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`.

Примечание: If you are accessing the userSettingsStore from within the component then you need to use `this.$app` instead `app`.

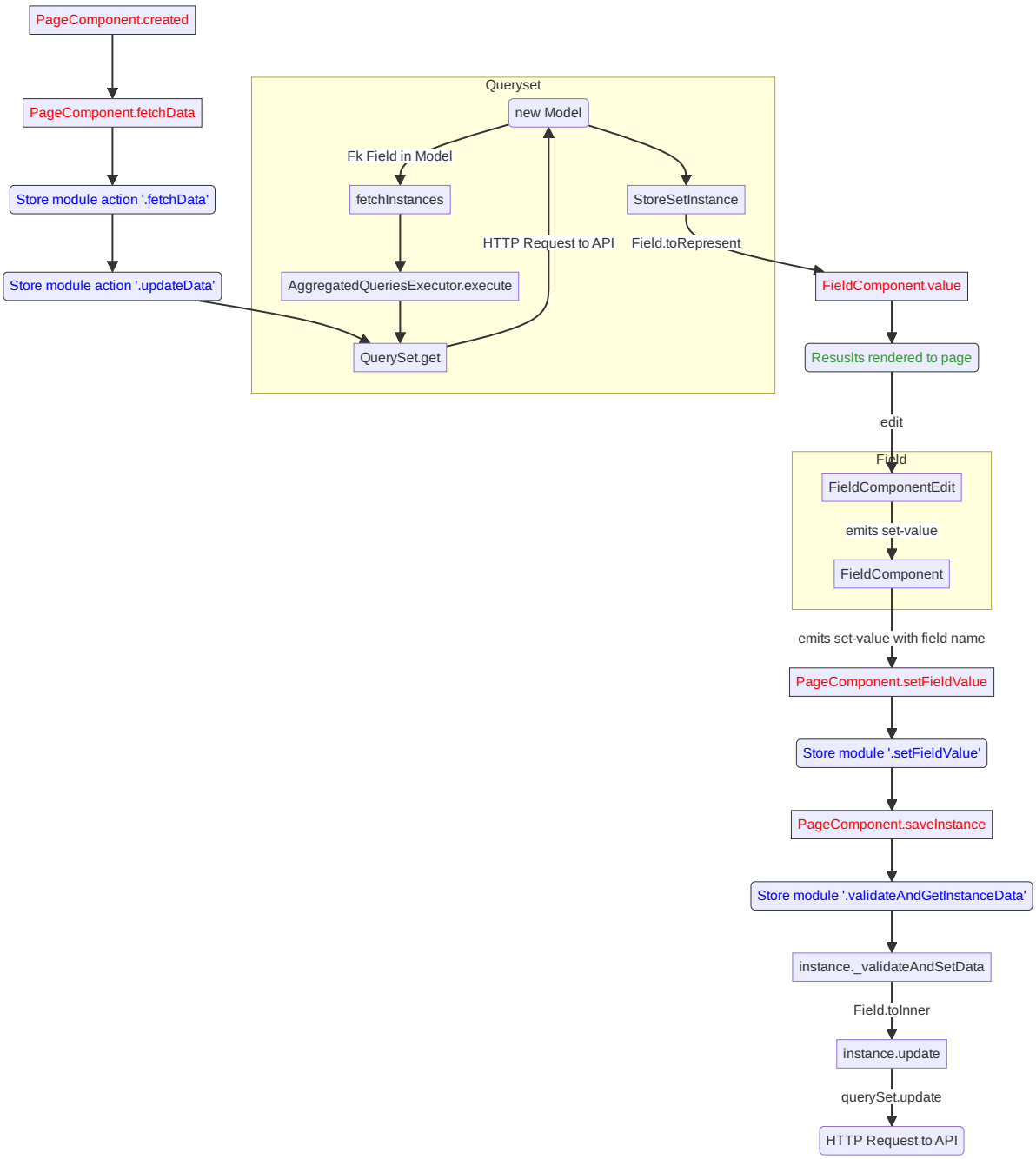
From *app.store* you may need:

- *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.

Содержание модулей Python

v

Алфавитный указатель

E

F

G

H

I

S

T

U

V

W