Developer documentation

Выпуск 5.9.0

VST Consulting

февр. 08, 2024

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

(continues on next page)

(продолжение с предыдущей страницы)

```
│   ├── 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;
- **pyproject.toml** - 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
   ```

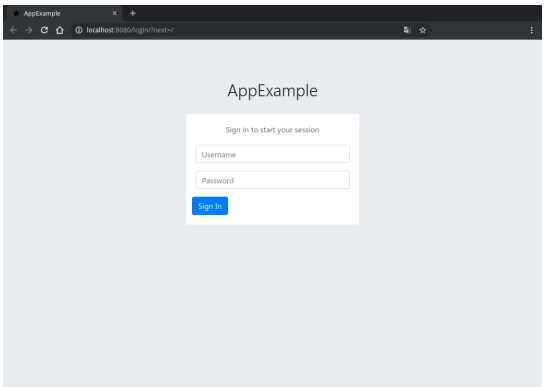
5. **Запуск приложения**

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

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

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` - 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. Используйте параметр `override_params` в тех случаях, когда необходимо перегрузить параметры генерируемого представления в качестве вложенного (работает только когда задан `model` как вложенное представление).
- `_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.
```

Метод `get_view_class()` — это служебный метод в ORM Django моделях, предназначенный для облегчения настройки и создания экземпляров представлений Django Rest Framework (DRF). Это позволяет разработчикам определить и настроить различные аспекты класса представления DRF.

Примеры:

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

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

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

Разработчики могут использовать этот метод для изменения различных аспектов получаемого представления, таких как классы сериализаторов, конфигурацию полей, фильтры,классы разрешений и т.п. Этот метод использует такие же атрибуты, которые были объявлены в мета-атрибутах, но позволяет перегружать отдельные части.

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*.

[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/

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`:

```
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
```

(continues on next page)

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

(продолжение с предыдущей страницы)

```
'''
@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'
)

# 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'
})
```

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

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):

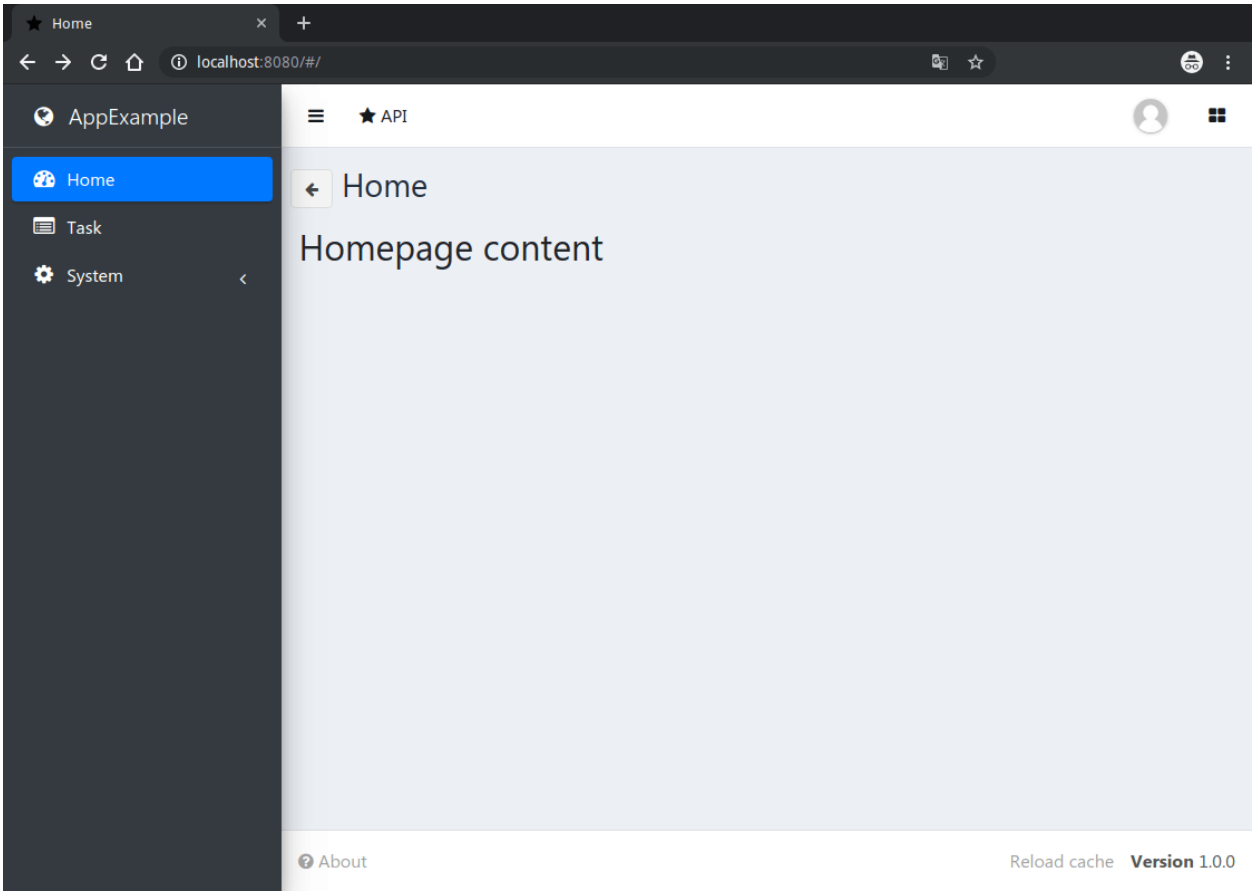
```
python -m {{app_name}} web stop=/tmp/{{app_name}}_web.pid
```

Затем запустите его снова:

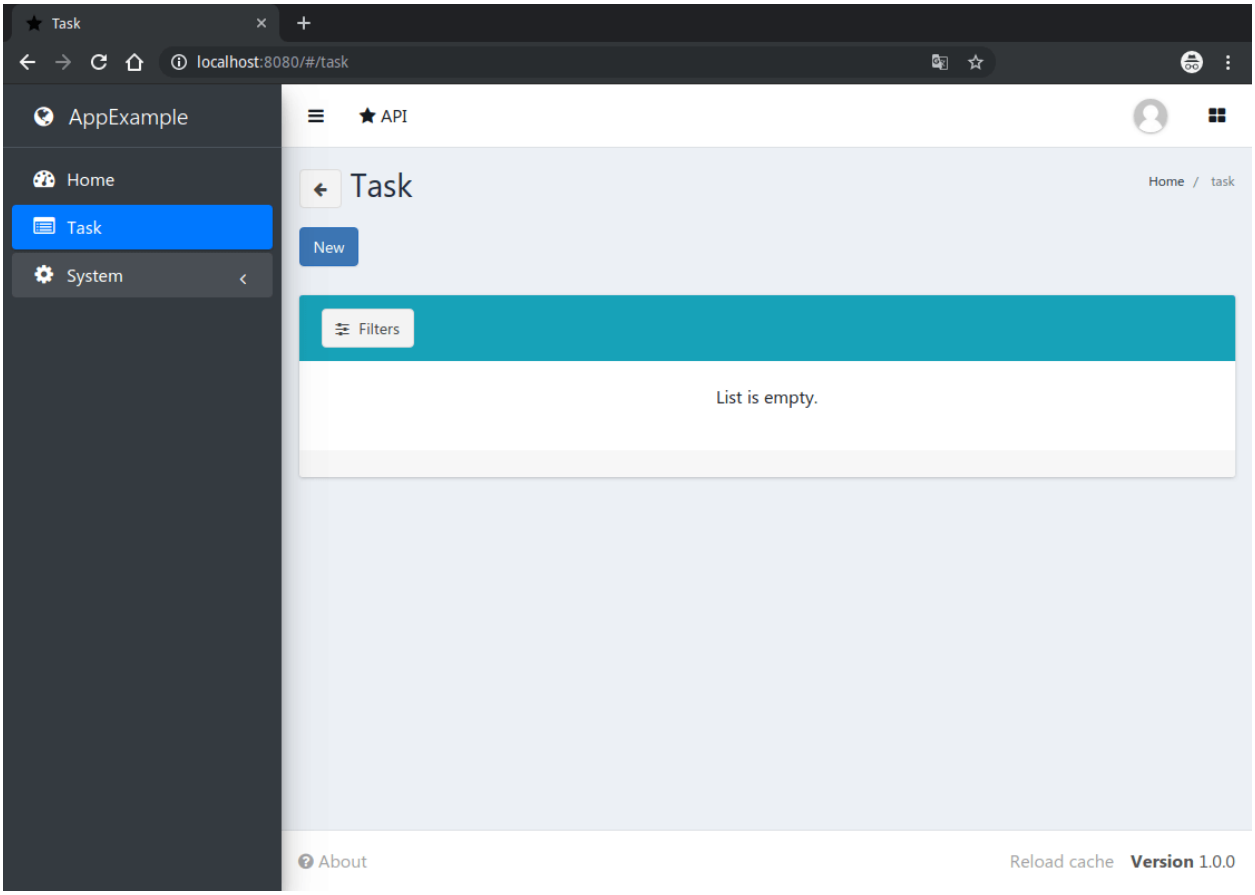
```
python -m {{app_name}} web
```

После перезагрузки кэша вы увидите следующую страницу:

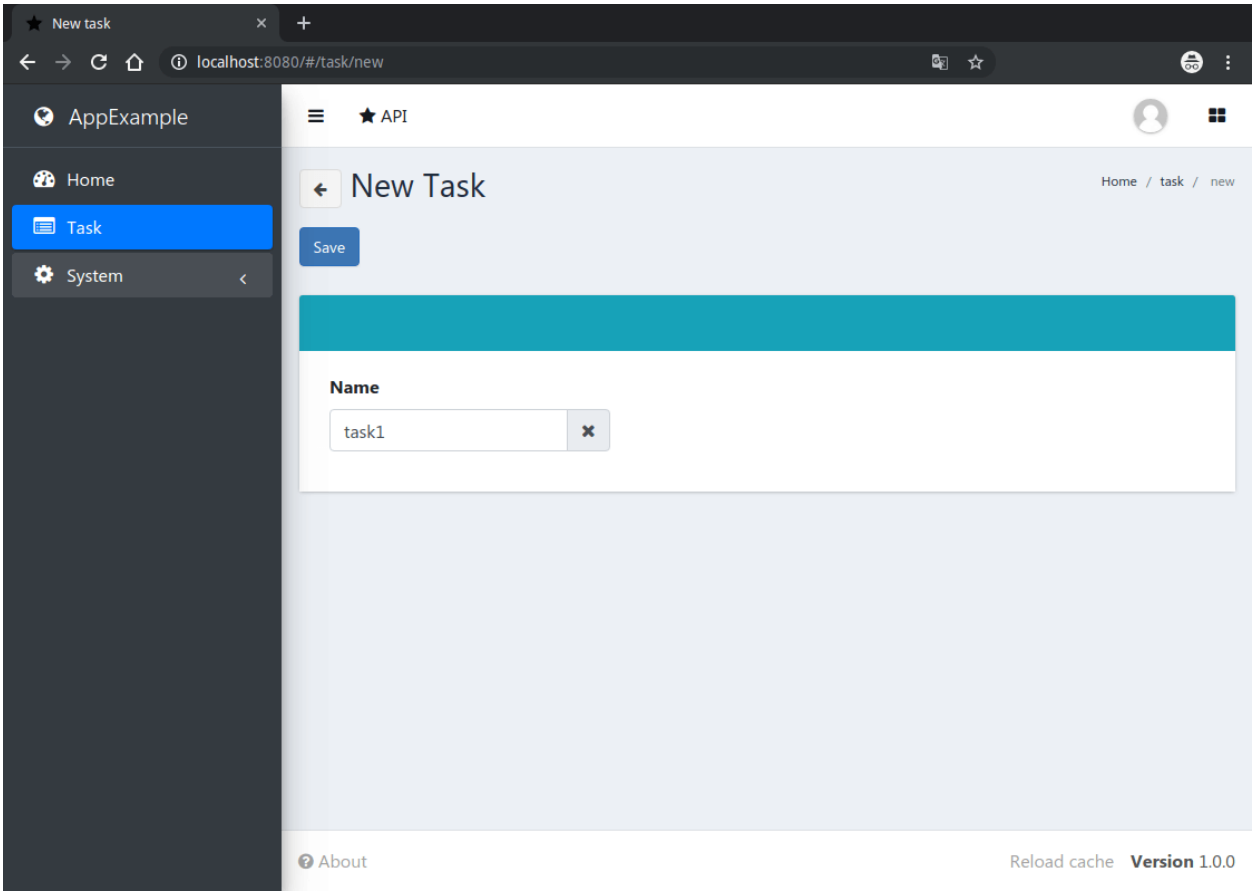
[7] https://docs.djangoproject.com/en/3.2/topics/migrations/



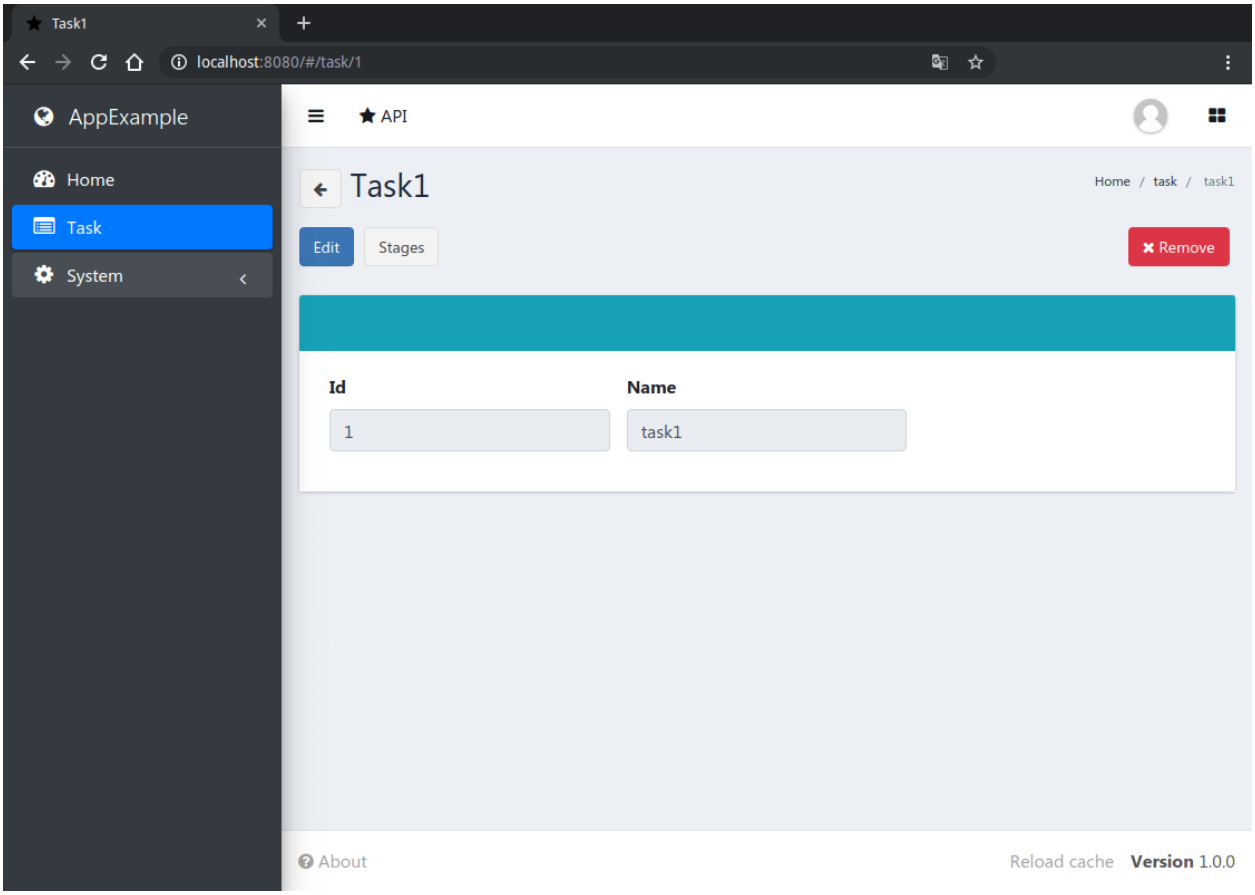
Как вы можете видеть, ссылка на новое Task view добавлена в боковое меню. Давайте нажмем на нее.



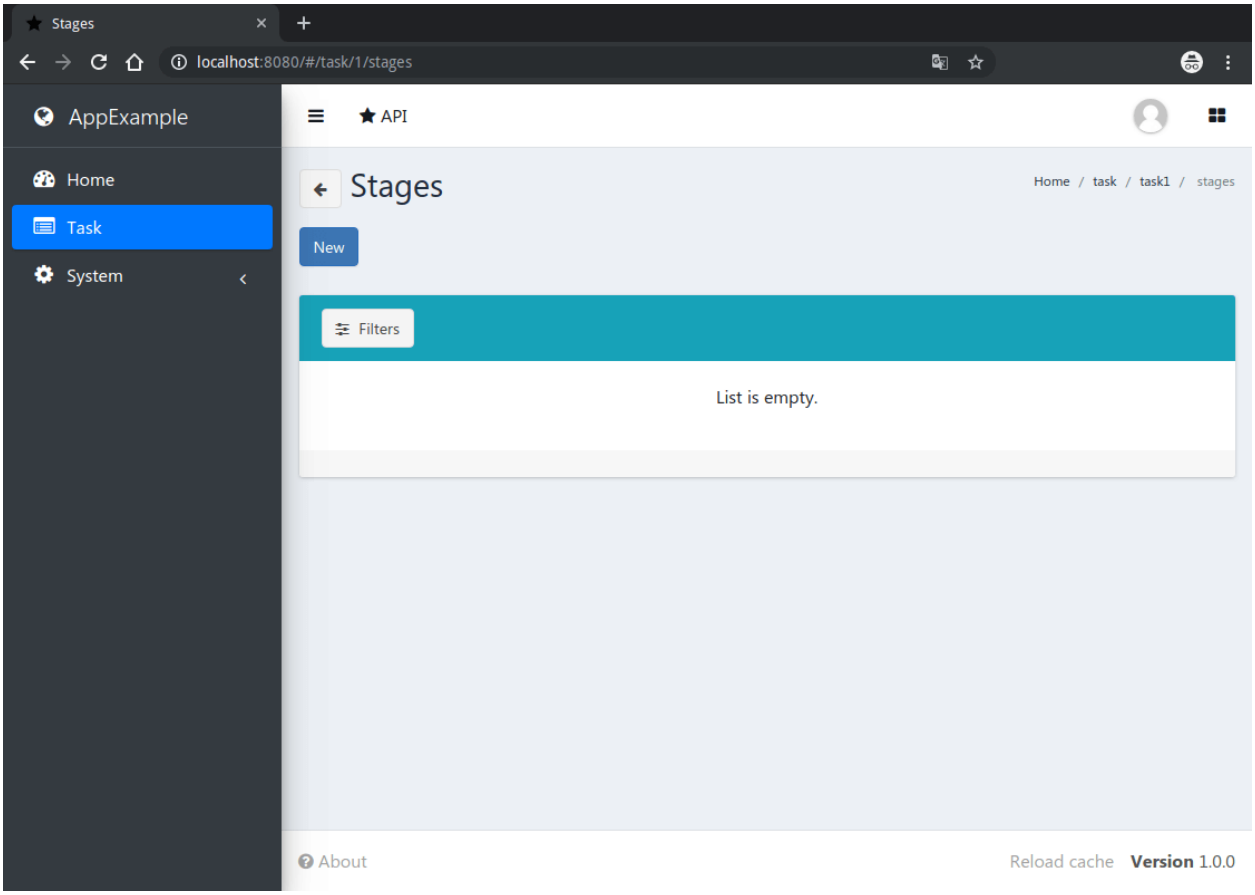
В вашем приложении нет экземпляра задачи. Добавьте его, используя кнопку „new“.



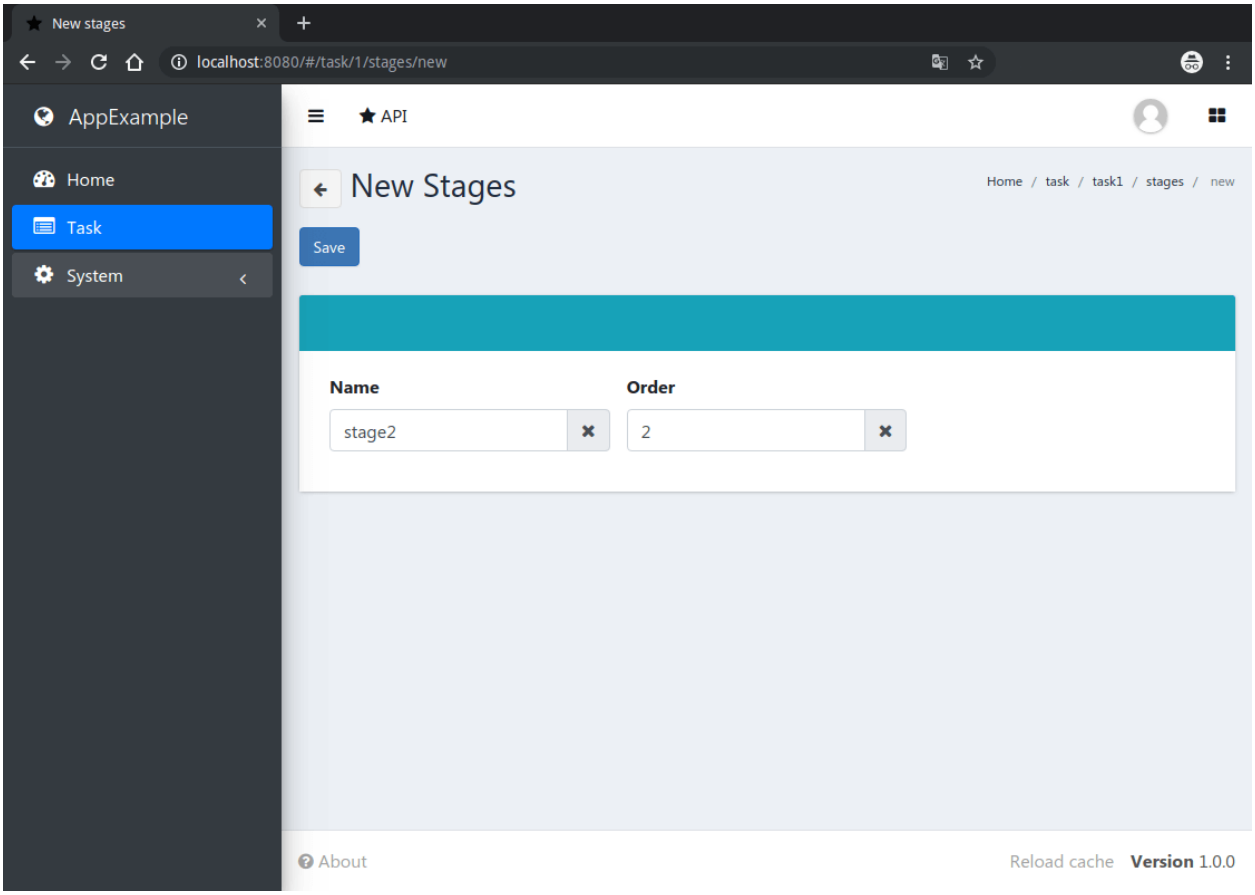
После создания новой задачи вы увидите следующую страницу:

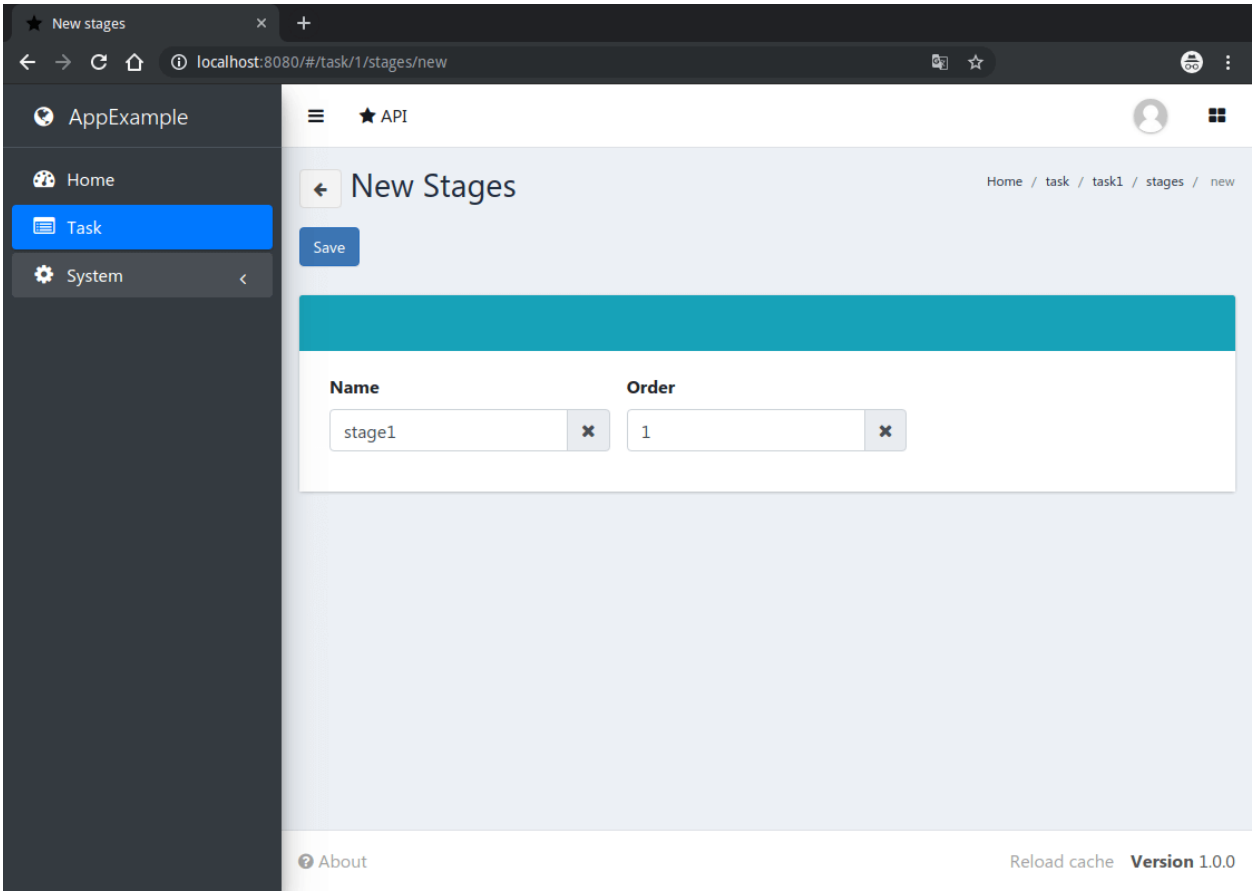


Как видите, есть кнопка „stages“, которая открывает страницу со списком этапов этой задачи. Давайте на нее нажмем.

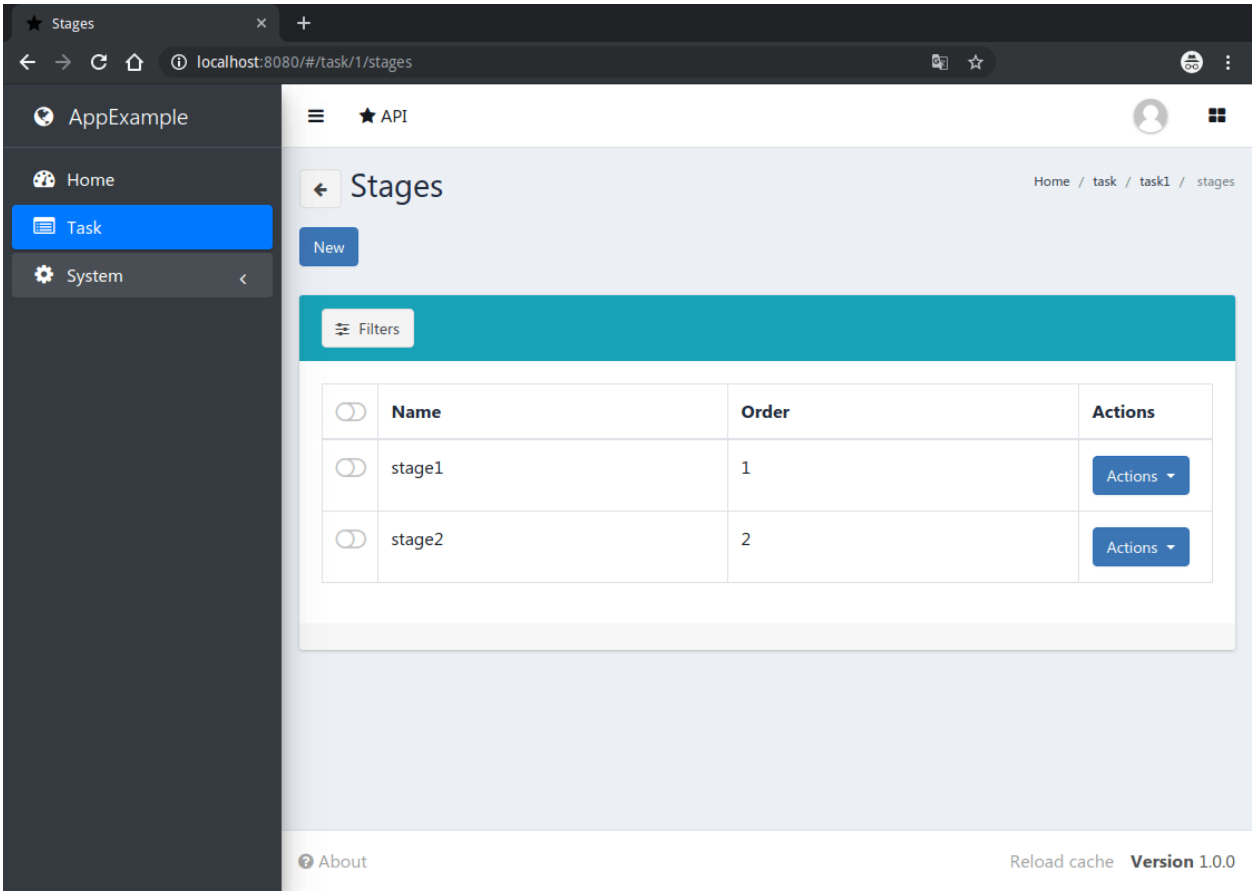


В вашем приложении нет экземпляров этапов. Давайте создадим 2 новых этапа.


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, Tarantool 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.К примеру: `ldap://your_ldap_server:389`
- **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 отображения логов. Этот уровень настраивается для Django и Celery, определяет объем информации в журнале. С более высокими уровнями предоставляет детальную информацию для целей отладки во время разработки и более низкие уровни выдают необходимую информацию для рабочих окружений. По умолчанию уровень: 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.

[10] https://docs.python.org/3.8/library/pickle.html#module-pickle
[11] https://en.wikipedia.org/wiki/Vigenèrecipher

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, которое используется когда соответствующий бэкенд это поддерживает. Табличное пространство — это место в файловой системе на сервере базы данных, где хранятся физические файлы данных, соответствующие таблицам базы данных. Это позволяет вам организовывать и управлять хранением таблиц вашей базы данных, указывая место на диске, где будут располагаться данные таблицы.
- **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[12].»
- **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[13]. 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[14]

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`:

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

```
[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
```

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[15]. При кластеризации мы рекомендуем делить кэш между узлами для повышения производительности с использованием реализаций клиент-серверного кэша. Мы рекомендуем использовать Redis в производственных окружениях.

2.4.1 Tarantool в качестве сервера кеша для Django

`TarantoolCache` это уникальный бэкенд для кеша Django, который позволяет использовать Tarantool как сервер-хранилище кешей. Чтобы использовать этот механизм необходимо выставить следующие настройки в конфигурации проекта:

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

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

Расшифровка настроек

- **location** - Имя хоста и порт для подключения к серверу Tarantool.
- **backend** - Путь до класса TarantoolCache бэкенда.
- **space** - Имя спейса в Tarantool для использования кешем (по умолчанию `DJANGO_CACHE`).
- **user** - Имя пользователя для подключения к Tarantool-серверу. (По умолчанию: `guest`).
- **password** - Пароль для подключения к серверу Tarantool. Не обязательный.

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

Можно дополнительно указать переменную `connect_on_start` в секции `[cache.options]`. Когда задано значение `true`, это указывает на вызов дополнительной инициализации на Tarantool сервере для настройки спейсов и настройки сервиса для автоматического удаления просроченных объектов.

Предупреждение: Обратите внимание, что это требует установки модуля `expirationd` на сервере Tarantool.

Примечание: Когда используется Tarantool в качестве кеша то, автоматически создаются временные (в памяти) спейсы для операций с кешем. Эти временные спейсы динамически создаются по необходимости и нужны, чтобы хранить временные данные максимально эффективно.

Это важно, что эти спейсы автоматически создаются в памяти, и если необходимо сделать, чтобы эти спейсы хранили данные на дисках, то необходимо заранее создать их на серевере Tarantool с такой же схемой и аналогичными настройками, которые используются в VST Utils. Убедитесь, что вам необходимо использовать устойчивые хранилища в вашем приложении, прежде чем создавать их на сервере Tarantool с аналогичными названиями. Это важно для стабильной и устойчивой работы.

Примечание: Важно отметить, что этот драйвер кеша уникальный для vstutils и адаптирован для полной интеграции с фреймворком VST Utils.

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 Tarantool, Redis или Memcached в качестве бэкэнда, потому что они имеют достаточную производительность для этих целей. 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]`.

Celery — это распределенная система очередей задач для обработки асинхронных задач в веб-приложениях. Его основная роль — облегчить выполнение фоновых или трудоемких задач независимо от основной логики приложения. Celery особенно полезен для разгрузки задач, которые не требуют немедленной обработки, что повышает общую скорость отклика и производительность приложения.

Ключевые функции и роли Celery в приложении с асинхронными задачами включают:

1. Асинхронное выполнение задач: Celery позволяет разработчикам определять задачи как функции или методы и выполнять их асинхронно. Это полезно для задач, которые могут занять значительное время, таких как отправка электронных писем, обработка данных или создание отчетов.
2. Распределенная архитектура: Celery работает распределенным образом, что делает его подходящим для крупномасштабных приложений. Он может распределять задачи между несколькими рабочими процессами или даже несколькими серверами, повышая масштабируемость и производительность.
3. Интеграция очереди сообщений: Celery полагается на брокеров сообщений (таких как RabbitMQ, Redis, Tarantool, SQS или другие) для управления связью между основным приложением и рабочими процессами. Такое разделение обеспечивает надежное выполнение задач и позволяет эффективно обрабатывать очереди задач.
4. Периодические задачи: Celery включает в себя планировщик, который позволяет выполнять периодические или повторяющиеся задачи. Это полезно для автоматизации задач, которые необходимо выполнять через определенные промежутки времени, например обновления данных или выполнения операций обслуживания.
5. Обработка ошибок и механизм повторных попыток: Celery предоставляет механизмы для обработки ошибок в задачах и поддерживает автоматические повторные попытки. Это обеспечивает надежность выполнения задач, позволяя системе восстанавливаться после временных сбоев.
6. Хранение результатов задач: Celery поддерживает хранение результатов выполненных задач, что может быть полезно для отслеживания хода выполнения задач или получения результатов позже. Эта функция особенно ценна для длительных задач.

Приложение, основанное на vstutils, использует Celery для выполнения длительных асинхронных задач. 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[16]. По умолчанию: `filesystem:///var/tmp`.
- **concurrency** - Количество потоков рабочего процесса Celery. По умолчанию: 4.
- **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[17] (с соответствующими типами):

- **prefetch_multiplier** - CELERYD_PREFETCH_MULTIPLIER[18]
- **max_tasks_per_child** - CELERYD_MAX_TASKS_PER_CHILD[19]

[16] http://docs.celeryproject.org/en/latest/userguide/configuration.html#conf-broker-settings
[17] http://docs.celeryproject.org/en/latest/userguide/configuration.html#new-lowercase-settings
[18] http://docs.celeryproject.org/en/latest/userguide/configuration.html#std-setting-worker_prefetch_multiplier
[19] http://docs.celeryproject.org/en/latest/userguide/configuration.html#std-setting-worker_max_tasks_per_child

- **results_expiry_days** - CELERY_RESULT_EXPIRES[20]
- **default_delivery_mode** - CELERY_DEFAULT_DELIVERY_MODE[21]
- **task_send_sent_event** - CELERY_DEFAULT_DELIVERY_MODE[22]
- **worker_send_task_events** - CELERY_DEFAULT_DELIVERY_MODE[23]

VST Utils так же предоставляет поддержку для использования Tarantool сервера как транспорта в Celery, обеспечивая эффективную и надежную передачу сообщений между распределёнными компонентами. Для использования этой возможности на сервере Tarantool должны быть установлены модули *queue* и *expirationd*.

Для настройки подключения используйте в качестве примера следующий URL: `tarantool://guest@localhost:3301/rpc`

- `tarantool://`: Указывает тип транспорта.
- *guest*`: Параметры аутентификации (в данном случае без пароля).
- `localhost`: Адрес сервера.
- `3301`: Порт для подключения.
- `rpc`: Префикс для имён очередей и/или спейса хранилища результата

VST Utils так же поддерживает Tarantool в качестве бэкенда для хранения результатов задач Celery. Строка подключения аналогична как и у транспорта.

Примечание: Когда Tarantool используется для хранения результатов или в качестве транспорта в VST Utils, то автоматически создаются спейсы и очереди для этих операций. Эти временные хранилища создаются динамически по мере необходимости и необходимы для эффективного хранения временных данных.

Это важно, что эти спейсы автоматически создаются в памяти, и если необходимо сделать, чтобы эти спейсы хранили данные на дисках, то необходимо заранее создать их на серевере Tarantool с такой же схемой и аналогичными настройками, которые используются в VST Utils. Убедитесь, что вам необходимо использовать устойчивые хранилища в вашем приложении, прежде чем создавать их на сервере Tarantool с аналогичными названиями. Это важно для стабильной и устойчивой работы.

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.

[20] http://docs.celeryproject.org/en/latest/userguide/configuration.html#std-setting-result_expires
[21] http://docs.celeryproject.org/en/latest/userguide/configuration.html#task-default-delivery-mode
[22] http://docs.celeryproject.org/en/latest/userguide/configuration.html#task_send_sent_event
[23] http://docs.celeryproject.org/en/latest/userguide/configuration.html#worker_send_task_events

- **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`. Эти два бэкэнда служат разным целям в разных средах. SMTP обеспечивает надежную доставку электронной почты в производственных условиях, а консольный вариант обеспечивает удобный способ проверки электронной почты во время разработки без риска непреднамеренного взаимодействия с внешними почтовыми серверами. Разработчики часто переключаются между этими бэкэндами в зависимости от контекста своей работы, выбирая тот, который подходит для этапа разработки или тестирования.

- **host** - IP-адрес или доменное имя smtp-сервера. Если не указано, vstutils использует бэкэнд `console`. По умолчанию: `None`.
- **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[24] 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`.

[24] https://github.com/adamchainz/django-cors-headers#configuration

- **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[25]. По умолчанию: `None`.
- **csrf_trusted_origins** - Список хостов, которым доверяются небезопасные запросы. Подробнее[26]. 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[27]. По умолчанию: `HTTP_X_FORWARDED_PROTOCOL`.
- **secure_proxy_ssl_header_value** - Значение заголовка, которое активирует использование URL-адресов SSL в ответах. Подробнее[28]. По умолчанию: `https`.

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

- **secure_browser_xss_filter** - SECURE_BROWSER_XSS_FILTER[29]
- **secure_content_type_nosniff** - SECURE_CONTENT_TYPE_NOSNIFF[30]
- **secure_hsts_include_subdomains** - SECURE_HSTS_INCLUDE_SUBDOMAINS[31]
- **secure_hsts_preload** - SECURE_HSTS_PRELOAD[32]
- **secure_hsts_seconds** - SECURE_HSTS_SECONDS[33]
- **password_reset_timeout_days** - PASSWORD_RESET_TIMEOUT_DAYS[34]
- **request_max_size** - DATA_UPLOAD_MAX_MEMORY_SIZE[35]
- **x_frame_options** - X_FRAME_OPTIONS[36]
- **use_x_forwarded_host** - USE_X_FORWARDED_HOST[37]
- **use_x_forwarded_port** - USE_X_FORWARDED_PORT[38]

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`.

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

- **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]`.

Centrifugo используется для оптимизации моментального обновления данных в приложении Django, обеспечивая беспрепятственное взаимодействие между его различными компонентами. Основной принцип работы заключается в оркестрированном создании сигналов Django, а именно `post_save` и `post_delete`, которые динамически вызываются во время HTTP-запросов или выполнения задач Celery. Эти сигналы, когда они вызываются на моделях пользователей или моделях, унаследованных от BaseModel в рамках фреймворка vstutils, инициируют создание сообщений для всех подписчиков, заинтересованных в событиях, связанных с этими моделями. После завершения HTTP-запроса или задачи Celery механизм уведомлений отправляет настроенные сообщения всем соответствующим подписчикам. Это означает, что каждая активная вкладка браузера с соответствующей подпиской мгновенно получает уведомление, стимулируя мгновенный запрос на обновление данных. Отсутствие необходимости для приложений в периодическом опросе REST API в фиксированные интервалы (например, каждые 5 секунд) существенно снижает операционную нагрузку REST API и обеспечивает его масштабируемость для более крупной базы пользователей. Важно отметить, что эту модель моментального обмена данными можно рассматривать как альтернативу по сравнению с периодическими запросами. Вместе с тем, она гарантирует мгновенное и синхронизированное обновление данных, способствуя высококачественному пользовательскому опыту.

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 с меткой модели и первичным ключом. Без этой службы все клиенты GUI получают данные страницы каждые 5 секунд (по умолчанию).

- **address** - Адрес сервера Centrifugo.
- **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[39] и Boto3[40].

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[41].

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[42].

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]` пусто, 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
```

(continues on next page)

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

(продолжение с предыдущей страницы)

```
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[43].

2.14 Настройки веб-уведомлений

Раздел `[webpuwsh]`.

- **enabled**: Булевый флаг, который включает или отключает веб-уведомления. Установите *true*, чтобы активировать веб-уведомления, и *false*, чтобы деактивировать их. По умолчанию: *false*. Если установлено значение false, то настройки уведомлений на странице пользователя будут скрыты, и метод *send* класса уведомлений ничего не будет делать.
- **vapid_private_key**, **vapid_public_key**: Это ключи сервера приложений, используемые для отправки push-уведомлений. Ключи VAPID (Voluntary Application Server Identification) состоят из публичного и приватного ключей. Эти ключи являются необходимыми для безопасной связи между вашим сервером и службой push. Для генерации пары ключей VAPID и понимания их использования смотрите подробное руководство, доступное здесь: Создание ключей VAPID[44].
- **vapid_admin_email**: Эта настройка указывает адрес электронной почты администратора или ответственного за сервер. Это контактная точка для службы push, чтобы связаться в случае каких-либо проблем или нарушений политики.

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

- **default_notification_icon**: URL изображения иконки по умолчанию, которая будет использоваться для веб-уведомлений. Для избежания путаницы предпочтительно использовать абсолютный URL. Эта иконка будет отображаться в уведомлениях, если на уровне уведомления не указана другая иконка. Более подробную информацию об иконке можно найти здесь[45].

Для более подробного руководства по использованию и реализации веб-уведомлений в VSTUtils смотрите руководство по веб-уведомлениям, *here*.

Помните, что эти настройки критически важны для правильной работы и надежности веб-уведомлений в вашем приложении. Убедитесь, что они настроены правильно для оптимальной производительности.

2.15 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.[46].

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

2.16 Работа за прокси-сервером с поддержкой TLS

2.16.1 Nginx

Чтобы настроить vstutils для работы через Nginx с поддержкой TLS, выполните следующие шаги:

1. **Установка Nginx:**

Убедитесь, что Nginx установлен на вашем сервере. Вы можете установить его с помощью менеджера пакетов, специфичного для вашей операционной системы.

2. **Настройка Nginx:**

Создайте файл конфигурации Nginx для вашего приложения vstutils. Ниже приведен базовый пример конфигурации Nginx. Измените значения в соответствии с вашей конкретной настройкой.

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

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

server {
    listen 443 ssl;
    server_name your_domain.com;

    ssl_certificate /path/to/your/certificate.crt;
    ssl_certificate_key /path/to/your/private.key;
    ssl_protocols TLSv1.2 TLSv1.3;
```

(continues on next page)

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

(продолжение с предыдущей страницы)

```
    ssl_ciphers 'TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_
↪SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384';

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

    charset utf-8;

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

Замените `your_domain.com` на ваш реальный домен и обновите пути к SSL-сертификатам.

3. **Обновление настроек vstutils:**

Убедитесь, что настройки vstutils содержат правильные конфигурации для HTTPS. В вашем `/etc/vstutils/settings.ini` (или в проекте `settings.ini`):

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

Это гарантирует, что vstutils распознает соединение по протоколу HTTPS.

4. **Перезапустите Nginx:**

После внесения этих изменений перезапустите Nginx, чтобы применить новые конфигурации:

```
sudo systemctl restart nginx
```

Теперь ваше приложение vstutils должно быть доступно через HTTPS через Nginx. Измените эти инструкции в зависимости от вашего конкретного окружения и требований к безопасности.

2.16.2 Traefik

Чтобы настроить vstutils для работы через Traefik с поддержкой TLS, выполните следующие шаги:

1. **Установка Traefik:**

Убедитесь, что Traefik установлен на вашем сервере. Вы можете скачать двоичный файл с официального сайта или использовать менеджер пакетов, специфичный для вашей операционной системы.

2. **Настройка Traefik:**

Создайте файл конфигурации Traefik `/path/to/traefik.toml`. Вот базовый пример:

3. **Создайте конфигурацию Traefik Toml:**

Создайте файл `/path/to/traefik_config.toml` с следующим содержимым:

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

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

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

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

Обязательно замените `your_domain.com` на ваш реальный домен.

4. **Обновление настроек vstutils:**

Убедитесь, что настройки vstutils содержат правильные конфигурации для HTTPS. В вашем `/etc/vstutils/settings.ini` (или в проекте `settings.ini`):

5. **Запустите Traefik:**

Запустите Traefik следующей командой:

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

Теперь ваше приложение vstutils должно быть доступно через HTTPS через Traefik. Измените эти инструкции в зависимости от вашего конкретного окружения и требований.

2.16.3 HAProxy

1. **Установка HAProxy:**

Убедитесь, что HAProxy установлен на вашем сервере. Вы можете установить его с помощью менеджера пакетов, специфичного для вашей операционной системы.

2. **Настройка HAProxy:**

Создайте файл конфигурации HAProxy для вашего приложения vstutils. Вот базовый пример конфигурации HAProxy. Измените значения в соответствии с вашей конкретной настройкой.

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

frontend https-in
    bind *:443 ssl crt /path/to/your/certificate.pem
    mode http
    option forwardfor
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    http-request set-header X-Forwarded-Proto https

    default_backend vstutils_backend

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

Замените `your_domain.com` на ваш реальный домен и обновите пути к SSL-сертификатам.

3. **Обновление настроек vstutils:**

Убедитесь, что настройки vstutils содержат правильные конфигурации для HTTPS. В вашем `/etc/vstutils/settings.ini` (или в проекте `settings.ini`):

4. **Перезапуск HAProxy:**

После внесения этих изменений перезапустите HAProxy, чтобы применить новые конфигурации.

```
sudo systemctl restart haproxy
```

Теперь ваше приложение vstutils должно быть доступно через HTTPS через HAProxy. Измените эти инструкции в зависимости от вашего конкретного окружения и требований к безопасности.

2.17 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.

В контексте vstutils внедрение `uvloop` играет ключевую роль в оптимизации производительности приложения, особенно при использовании `uvicorn` в качестве ASGI-сервера. `uvloop` представляет собой ультра-быстрый, готовый к использованию заменитель стандартного цикла событий, предоставляемого Python. Он построен на базе `libuv`, библиотеки высокопроизводительного цикла событий, и специально разработан для оптимизации скорости выполнения асинхронного кода.

Используя `uvloop`, разработчики могут достичь значительного улучшения производительности за счет снижения задержек и увеличения пропускной способности. Это особенно важно в сценариях, где приложения обрабатывают большое количество одновременных подключений. Улучшенная эффективность обработки цикла событий напрямую переводится в более быстрое время ответа и лучшую общую отзывчивость приложения.

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. Используйте параметр `override_params` в тех случаях, когда необходимо перегрузить параметры генерируемого представления в качестве вложенного (работает только когда задан `model` как вложенное представление).
- `_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[47] 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.
```

Метод `get_view_class()` — это служебный метод в ORM Django моделях, предназначенный для облегчения настройки и создания экземпляров представлений Django Rest Framework (DRF). Это позволяет разработчикам определить и настроить различные аспекты класса представления DRF.

Примеры:

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

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
        },
    },
)
```

Разработчики могут использовать этот метод для изменения различных аспектов получаемого представления, таких как классы сериализаторов, конфигурацию полей, фильтры,классы разрешений и т.п. Этот метод использует такие же атрибуты, которые были объявлены в мета-атрибутах, но позволяет перегружать отдельные части.

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.

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 для превращения методов модели в сгенерированные view экшены[48]. Когда метод декорируется, он становится частью сгенерированного view, и ссылка *self* внутри метода указывает на объект view. Это позволяет расширять функциональность сгенерированных view с помощью пользовательских экшенов.

Декоратор *register_view_action* поддерживает различные аргументы, и вы можете обратиться к документации для *vstutils.api.decorators.subaction()*, чтобы изучить полный список поддерживаемых аргументов. Эти аргументы предоставляют гибкость в определении поведения и характеристик сгенерированных экшенов view.

Примечание: В сценариях, где вы работаете с прокси-моделями, использующими общий набор действий, вы можете использовать именованный аргумент *inherit* со значением *True*. Это позволяет прокси-модели наследовать действия, определенные в базовой модели, сокращая избыточность и способствуя повторному использованию кода.

---

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

Примечание: Во многих случаях действие может не требовать параметров и может быть выполнено, отправив пустой запрос. Для упрощения разработки и повышения эффективности декоратор *register_view_action* устанавливает сериализатор по умолчанию на *vstutils.api.serializers.EmptySerializer*. Это означает, что действие не ожидает входных данных, что удобно для действий, которые работают без дополнительных параметров.

Пример:

В этом примере показано, как использовать декоратор для создания пользовательского действия в представлении модели. Метод `empty_action` становится частью сгенерированного view и не ожидает входных параметров.

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


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

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

Vstutils поддерживает модели, которые не требуют прямого взаимодействия с базой данных или не являются непосредственно таблицами в базе. Эти модели проявляют разнообразные поведения, такие как извлечение данных непосредственно из атрибутов класса, загрузка данных из файлов или реализация собственных механизмов получения данных. Замечательно, что существуют модели, которые, в каком-то смысле, реализуют механизм SQL представлений с предопределенными запросами. Эта гибкость позволяет разработчикам определять широкий спектр моделей, от моделей, существующих только в памяти, до тех, которые без проблем интегрируют внешние источники данных. Система моделей Vstutils не ограничивается традиционными структурами, поддерживаемыми базой данных, предоставляя гибкое основание для создания различных представлений данных.

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

Представляет собой кастомную модель, предназначенную для самостоятельной реализации запросов ко внешним сервисам.

Данная кастомная модель облегчает взаимодействие с внешними сервисами, позволяя передавать параметры фильтрации, лимитирования и сортировки во внешний запрос. Она предназначена для получения данных, которые уже отфильтрованы и ограничены.

Для эффективного использования этой модели разработчики должны реализовать метод класса `get_data_generator()`. Этот метод получает объект запроса с необходимыми параметрами, позволяя разработчикам настраивать взаимодействие с внешними сервисами.

Пример:

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

    class Meta:
        managed = False
```

(continues on next page)

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

(продолжение с предыдущей страницы)

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

classmethod get_data_generator(*query*)

Этот метод класса должен быть реализован в производных классах для определения пользовательской логики извлечения данных из внешнего сервиса на основе предоставленных параметров запроса.

Объект запроса может содержать следующие параметры:

- filter (dict): Словарь, задающий критерии фильтрации.
- exclude (dict): Словарь, задающий критерии исключения.
- order_by (list): Список, задающий порядок сортировки.
- low_mark (int): Нижний индекс для среза (если задан срез).
- high_mark (int): Верхний индекс для среза (если задан срез).
- is_sliced (bool): Булево значение, указывающее, является ли запрос срезом.

Параметры
query (*dict*[49]) – Объект, содержащий параметры фильтрации, лимитирования и сортировки.

Результат
Генератор, возвращающий запрошенные данные.

Тип результата
Generator

Исключение
NotImplementedError[50] – Если метод не реализован в производном классе.

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

Кастомная модель, загружающая данные из YAML-файла вместо базы данных. Путь к файлу указывается в атрибуте *FileModel.file_path*.

Примеры:
Предположим, что исходный файл хранится в */etc/authors.yaml* со следующим содержимым:

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

Вы можете создать кастомную модель, используя этот файл:

```
from vstutils.custom_model import FileModel, CharField

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

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

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

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.

Переменные

data (*list*[51]) – Список кортежей данных. Пустой по умолчанию.

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

Реализует механизм программирования SQL View над другими моделями.

Эта модель предоставляет механизм для реализации поведения, аналогичного SQL View, над другими моделями. В методе `get_view_queryset()` подготавливается базовый запрос, и все последующие действия реализуются поверх него.

Примеры:

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

    class Meta:
        managed = False

    @classmethod
```

(continues on next page)

[51] https://docs.python.org/3.8/library/stdtypes.html#list

(продолжение с предыдущей страницы)

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

classmethod get_view_queryset()

Этот метод класса должен быть реализован в производных классах для определения пользовательской логики создания базового queryset для SQL View.

Результат
Базовый queryset для SQL View.

Тип результата
django.db.models.query.QuerySet[52]

Исключение
NotImplementedError[53] – Если метод не реализован в производном классе.

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`[54]. 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`[55]

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

Расширяет класс `django.db.models.TextField`[56]. Простое поле для хранения HTML-разметки. Поле основано на базе `django.db.models.TextField`[57], 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`[58]. 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:

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

```
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`[59] 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.

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`[60]. 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`[61]. Используйте это поле в *vstutils.models.BModel*, чтобы получить *vstutils.api.MultipleNamedBinaryImageInJSONField* в сериализаторе.

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

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

Расширяет `django.db.models.TextField`[62]. 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`[63]. 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`[64]. Простое поле для хранения строк в формате Markdown. Поле основано на `django.db.models.TextField`[65], 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.

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 сериализатора, обеспечивающее автодополнение на фронтенде с использованием указанного списка объектов.

Параметры

- **autocomplete** (*list*[66], *tuple*[67], *str*[68]) – Ссылка для автодополнения. Можно установить список или кортеж с значениями или указать имя определения схемы OpenAPI. Для имени определения, GUI найдет оптимальную ссылку и отобразит значения на основе аргументов `autocomplete_property` и `autocomplete_represent`.
- **autocomplete_property** (*str*[69]) – Указывает, какой атрибут из модели определения схемы OpenAPI использовать в качестве значения. По умолчанию „id“.
- **autocomplete_represent** – Указывает, какой атрибут из модели определения схемы OpenAPI использовать в качестве представленного значения. По умолчанию „name“.
- **use_prefetch** (*bool*[70]) – Загружать значения на фронтенде в режиме просмотра списка. Значение по умолчанию — `True`.

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

Примечание: Эта функциональность работает только в графическом интерфейсе. В API она ведет себя так же, как и *VSTCharField*.

Использование:

Это поле предназначено для использования в сериализаторах, где пользователь должен ввести значение, и требуется автодополнение на основе предопределенного списка или определения схемы OpenAPI. Если указана схема OpenAPI, два дополнительных параметра, `autocomplete_property` и `autocomplete_represent`, могут быть настроены для настройки внешнего вида выпадающего списка.

Пример:

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


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

# or

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

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

Поле для представления данных в виде штрихкода (Code 128) в пользовательском интерфейсе.

Это поле принимает и проверяет данные в виде допустимой ASCII-строки. Оно предназначено для отображения данных в виде штрихкода Code 128 в графическом пользовательском интерфейсе. Основные данные сериализуются или десериализуются с использованием указанного дочернего поля.

Параметры

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

Пример:

Предположим, у вас есть модель с полем *product_code*, и вы хотите отображать его представление в виде штрихкода Code 128 в графическом интерфейсе пользователя. Вы можете использовать *Barcode128Field* в своем сериализаторе:

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

class ProductSerializer(VSTSerializer):
```

(continues on next page)

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

(продолжение с предыдущей страницы)

```
    barcode = Barcode128Field(child=serializers.CharField(source='product_code
↪'))

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

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

Это поле расширяет функциональность *FileInStringField* и специально предназначено для обработки бинарных файлов. В интерфейсе пользователя оно выступает в качестве поля для загрузки файлов, принимая бинарные файлы от пользователя, которые затем конвертируются в строку в формате base64 и сохраняются в данном поле.

Параметры

media_types (*tuple*[71], *list*[72]) – Список MIME-типов, доступных для выбора пользователем. Поддерживается синтаксис с использованием `*`. По умолчанию `['*/*']`

Примечание: Эта функциональность работает только в графическом интерфейсе. В API она ведет себя так же, как и *VSTCharField*.

---

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

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

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*[73]) – Минимальная ширина ячейки. По умолчанию 200 px.
- **delimiter** (*str*[74]) – Символ-разделитель.
- **lineterminator** (*str*[75]) – Последовательность символов новой строки. Оставьте пустым для выбора автоматически. Возможные значения: `\r`, `\n`, `\r\n`.
- **quotechar** (*str*[76]) – Символ, используемый в качестве кавычек для полей.
- **escapechar** (*str*[77]) – Символ, используемый для экранирования кавычки в поле.
- **media_types** (*tuple*[78], *list*[79]) – Список MIME-типов, доступных для выбора пользователем. Поддерживается синтаксис с использованием `*`. По умолчанию `['text/csv']`

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

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

Поле, позволяющее пользователям вводить несколько значений, разделенных указанным разделителем (по умолчанию «,»). Извлекает список значений из другой модели или пользовательского списка и предоставляет автодополнение аналогично *AutoCompletionField*. Это поле подходит для полей-свойств модели, где основная логика уже реализована, или для использования с `model.CharField`.

Параметры

- **select** (*str*[80], *tuple*[81], *list*[82]) – Имя определения схемы OpenAPI или список со значениями.
- **select_separator** (*str*[83]) – Разделитель значений. По умолчанию - запятая.
- **select_represent** (*select_property,*) – Эти параметры работают аналогично `autocomplete_property` и `autocomplete_represent`. По умолчанию - `name`.
- **use_prefetch** (*bool*[84]) – Загружать значения на фронтенде в режиме просмотра списка. Значение по умолчанию - `False`.
- **make_link** (*bool*[85]) – Отображать значения как ссылки на модель. По умолчанию - `True`.
- **dependence** (*dict*[86]) – 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, и поле будет отключено.

Пример:

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

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

# or

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

Примечание: Эта функциональность работает только в графическом интерфейсе. В API она ведет себя так же, как и *VSTCharField*.

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

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*, специально разработанный для иерархических отношений на фронтенде.

Это поле предназначено для работы с отношениями ForeignKey в иерархической или древовидной структуре. Оно отображается в виде дерева на фронтенде, предоставляя четкое визуальное представление отношений родитель-ребенок.

Предупреждение: Это поле специально не поддерживает параметр `dependence`, так как работает в структуре дерева. Использование параметра `filters` следует приглядеться с осторожностью, поскольку неправильные фильтры могут нарушить иерархию дерева.

Параметры

- **only_last_child** (*bool*[87]) – Если True, поле ограничит выбор узлов без детей. Значение по умолчанию - *False*. Полезно, когда вы хотите обеспечить выбор листовых узлов.
- **parent_field_name** (*str*[88]) – Имя поля родителя в связанной модели. Значение по умолчанию - *parent*. Должно быть установлено в поле ForeignKey в связанной модели, представляющее отношения родитель-ребенок. Например, если у вашей связанной модели есть ForeignKey, например, *parent = models.ForeignKey(„self“, …)*, то *parent_field_name* должно быть установлено в *„parent“*.

Примеры:

Предположим, у вас есть связанная модель с полем ForeignKey, представляющим отношения родитель-ребенок:

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

Чтобы использовать DeepFkField с этой связанной моделью в сериализаторе, вы установили бы *parent_field_name* в „parent“:

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

В этом примере предполагается, что у вас есть связанная модель Category с полем ForeignKey „parent“. Затем DeepFkField отобразит категории в виде дерева на фронтенде, предоставляя интуитивно понятный механизм выбора для иерархических отношений.

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

---

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`[89] в модели или для экшенов.

Параметры

- **field** (*str*[90]) – 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*[91]) – 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*)

Специализированное поле, расширяющее *DynamicJsonTypeField* и проверяющее данные поля на основе `field_attribute`, выбранного в связанной модели. Данные поля проверяются на соответствие типу, определенному выбранным значением `field_attribute`.

Примечание: По умолчанию любое значение `field_attribute` проверяется как *VSTCharField*. Чтобы изменить это поведение, установите атрибут класса `{field_attribute}_fields_mapping` в связанной модели. Атрибут должен быть словарем, где ключи - это строковые представления значений `field_attribute`, а значения - экземпляры `rest_framework.Field` для проверки. Если значение не найдено в словаре, то тип по умолчанию будет *VSTCharField*.

---

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

Параметры

- **field** (*str*[92]) – Поле в модели, значение которого определяет тип текущего значения. Поле должно быть типа *FkModelField*.
- **field_attribute** (*str*[93]) – Атрибут экземпляра связанной модели, содержащий имя типа.
- **types** (*dict*[94]) – Отображение ключ-значение, где ключ - это значение подписанного поля, а значение - тип (в формате OpenAPI) текущего поля.

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

Пример:

Предположим, у вас есть модель с полем ForeignKey *related_model* и полем *type_attribute* в *RelatedModel*, которое определяет тип данных. Вы можете использовать *DependFromFkField* для динамической адаптации сериализации на основе значения *type_attribute*:

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

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

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

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

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

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

Универсальное поле сериализатора, которое динамически адаптирует свой тип в зависимости от значения другого поля в модели. Оно облегчает сложные сценарии, где тип данных для сериализации зависит от значения связанного поля.

Параметры

- **field** (*str*[95]) – Поле в модели, изменение значения которого будет динамически определять тип текущего поля.

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

- **types** (*dict*[96]) – Отображение ключ-значение, где ключ - это значение подписанного поля, а значение - тип (в формате OpenAPI) текущего поля.
- **choices** (*dict*[97]) – Варианты выбора для разных значений подписанных полей. Использует отображение, где ключ - это значение подписанного поля, а значение - список значений для выбора.
- **source_view** (*str*[98]) – Позволяет использовать данные родительских представлений в качестве источника для создания поля. Можно указать точный путь представления (*/user/{id}/*) или относительный указатель родителя (*<<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 детальные представления.

Пример:

Предположим, у вас есть сериализатор *MySerializer* с полем *field_type* (например, *ChoiceField*) и соответствующим полем *object_data*. Поле *object_data* может иметь разные типы в зависимости от значения *field_type*. Вот пример конфигурации:

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

В этом примере поле *object_data* динамически адаптирует свой тип в зависимости от выбранного значения *field_type*. Аргумент *types* определяет разные типы для каждого возможного значения *field_type*, позволяя гибкую и динамичную сериализацию.

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*[99], *list*[100]) – 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, предназначенная для использования в сериализаторах. Это поле позволяет

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

указать, какое поле связанной модели будет храниться в этом поле (по умолчанию: «id»), а какое поле будет представлять значение на фронтенде.

Параметры

- **select** (*str*[101]) – Имя определения схемы OpenAPI.
- **autocomplete_property** (*str*[102]) – Указывает, какой атрибут из модели определения схемы OpenAPI использовать в качестве значения. По умолчанию „id“.
- **autocomplete_represent** – Указывает, какой атрибут из модели определения схемы OpenAPI использовать в качестве представленного значения. По умолчанию „name“.
- **field_type** (*type*[103]) – Определяет тип поля autocomplete_property для дальнейшего описания в схеме и преобразования этого типа из API. По умолчанию пропускается, но требует объекты *int* или *str*.
- **use_prefetch** (*bool*[104]) – Загружать значения на фронтенде в режиме просмотра списка. Значение по умолчанию — `True`.
- **make_link** (*bool*[105]) – Отображать значение как ссылку на модель. Значение по умолчанию — `True`.
- **dependence** (*dict*[106]) – 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 будет отключено. Есть несколько специальных 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 выполнит запрос на фронтенде `/category?my_filter=3`, где `3` - первичный ключ текущего экземпляра.

Параметры

filters (*dict*[107]) – Словарь, где ключи - это имена полей из связанной модели (указанной в этом FkField), а значения - значения этого поля.

Примечание: Пересечение *dependence.values()* и *filters.keys()* вызовет ошибку для предотвращения неоднозначной 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`[108] 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*[109]) – 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`.

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*)

Специализированное поле для обработки HTML-текстового контента, отмеченного форматом: html.

Примечание: Это поле предназначено для использования в графическом интерфейсе пользователя (GUI) и работает аналогично *VSTCharField* в API.

Пример:

Если у вас есть модель с полем *html_content*, в котором хранится HTML-форматированный текст, вы можете использовать *HtmlField* в своем сериализаторе для обработки этого контента в графическом интерфейсе пользователя:

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

(continues on next page)

[101] https://docs.python.org/3.8/library/stdtypes.html#str
[102] https://docs.python.org/3.8/library/stdtypes.html#str
[103] https://docs.python.org/3.8/library/functions.html#type
[104] https://docs.python.org/3.8/library/functions.html#bool
[105] https://docs.python.org/3.8/library/functions.html#bool
[106] https://docs.python.org/3.8/library/stdtypes.html#dict
[107] https://docs.python.org/3.8/library/stdtypes.html#dict
[108] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.ForeignKey
[109] https://docs.python.org/3.8/library/stdtypes.html#str

(продолжение с предыдущей страницы)

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

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

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

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

Это поле предназначено для применения маски к значению на фронтенде. Оно расширяет класс „rest_framework.serializers.CharField“ и позволяет использовать библиотеку IMask[110] для определения масок.

Параметры

mask (*dict*[111], *str*[112]) – Маска, которая будет применена к значению. Это может быть как словарь, так и строка в формате библиотеки *IMask*.

Пример:

В сериализаторе включите это поле для применения маски к значению:

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

В этом примере предполагается, что сериализатор имеет поле *masked_value*, представляющее значение с предопределенной маской. *MaskedField* применит указанную маску на фронтенде, предоставляя маскированный ввод для пользователей.

Примечание: Эффективность этого поля ограничивается фронтендом, и маска применяется при вводе пользователя.

---

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*[113]) – 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.

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

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

Поле, представляющее бинарный файл в формате JSON.

Параметры

- **file** (*bool*[114]) – Если True, принимает только подклассы File в качестве входных данных. Если False, принимает только строковые входные данные. По умолчанию: False.
- **post_handlers** (*tuple*[115], *list*[116]) – Функции для обработки файла после валидации. Каждая функция принимает два аргумента: `binary_data` (байты файла) и `original_data` (ссылка на исходный JSON-объект). Функция должна возвращать обработанный `binary_data`.
- **pre_handlers** (*tuple*[117], *list*[118]) – Функции для обработки файла перед валидацией. Каждая функция принимает два аргумента: `binary_data` (байты файла) и `original_data` (ссылка на исходный JSON-объект). Функция должна возвращать обработанный `binary_data`.
- **max_content_size** (*int*[119]) – Максимально допустимый размер содержимого файла в байтах.
- **min_content_size** (*int*[120]) – Минимально допустимый размер содержимого файла в байтах.
- **min_length** (*int*[121]) – Минимальная длина имени файла. Применяется только при `file=True`.
- **max_length** (*int*[122]) – Максимальная длина имени файла. Применяется только при `file=True`.

Это поле предназначено для хранения бинарных файлов вместе с их именами в полях модели `django.db.models.CharField`[123] или `django.db.models.TextField`[124]. Все манипуляции, связанные с декодированием и кодированием данных бинарного содержимого, выполняются на клиенте, что 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`[125], если атрибут *file* установлен в значение *True*.

Пример:

В сериализаторе включите это поле для обработки бинарных файлов:

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

В этом примере предполагается, что в сериализаторе поле `binary_data` представляет информацию о бинарном файле в формате JSON. Поле `NamedBinaryFileInJsonField` обеспечит обработку хранения и извлечения бинарных файлов удобным для пользователя образом.

Бинарный файл представлен в формате JSON со следующими свойствами:

- **name** (str): Имя файла.
- **mediaType** (str): MIME-тип файла.
- **content** (str или File): Содержимое файла. Если *file* установлен в True, это будет ссылка на файл; в противном случае содержимое будет закодировано base64.

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

Предупреждение: Приложение клиента отобразит содержимое в виде ссылки для скачивания. Пользователи будут взаимодействовать с бинарным файлом через приложение, а обмен между Rest API и клиентом будет происходить через представленный объект JSON.

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

Поле, представляющее изображение в формате JSON, расширяющее *NamedBinaryFileInJsonField*.

Параметры

background_fill_color (*str*[126]) – 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.

Это поле предназначено для хранения бинарных файлов вместе с их именами в полях модели django.db.models.CharField[127] или django.db.models.TextField[128]. Оно расширяет возможности *NamedBinaryFileInJsonField* для специальной обработки изображений.

Кроме того, это поле проверяет изображение с использованием следующих валидаторов: - *vstutils.api.validators.ImageValidator* - *vstutils.api.validators.ImageResolutionValidator* - *vstutils.api.validators.ImageHeightValidator*

При сохранении и с добавленными валидаторами поле будет отображать соответствующее окно для настройки изображения по указанным параметрам, предоставляя удобный интерфейс для управления изображениями.

Бинарный файл представлен в формате JSON со следующими свойствами:

- **name** (str): Имя файла изображения.
- **mediaType** (str): MIME-тип файла изображения.
- **content** (str или File): Содержимое файла изображения. Если *file* установлен в True, это будет ссылка на файл; в противном случае содержимое будет закодировано base64.

Предупреждение: Приложение клиента отобразит содержимое как изображение. Пользователи будут взаимодействовать с изображением через приложение, и обмен между Rest API и клиентом будет происходить через представленный объект JSON.

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

Расширяет CharField[129], но в схеме имеет format = *password*. В пользовательском интерфейсе отображает все символы как звездочки вместо реально введенных данных.

[114] https://docs.python.org/3.8/library/functions.html#bool
[115] https://docs.python.org/3.8/library/stdtypes.html#tuple
[116] https://docs.python.org/3.8/library/stdtypes.html#list
[117] https://docs.python.org/3.8/library/stdtypes.html#tuple
[118] https://docs.python.org/3.8/library/stdtypes.html#list
[119] https://docs.python.org/3.8/library/functions.html#int
[120] https://docs.python.org/3.8/library/functions.html#int
[121] https://docs.python.org/3.8/library/functions.html#int
[122] https://docs.python.org/3.8/library/functions.html#int
[123] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.CharField
[124] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[125] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.FileField
[126] https://docs.python.org/3.8/library/stdtypes.html#str
[127] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.CharField
[128] https://docs.djangoproject.com/en/4.2/ref/models/fields/#django.db.models.TextField
[129] https://www.django-rest-framework.org/api-guide/fields/#charfield

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

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

Это поле предназначено для захвата и валидации номеров телефонов в международном формате. Оно расширяет класс „rest_framework.serializers.CharField“ и добавляет пользовательскую валидацию для обеспечения того, что номер телефона содержит только цифры.

Пример:

В сериализаторе включите это поле для обработки номеров телефонов:

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

В этом примере предполагается, что сериализатор имеет поле *phone_number*, представляющее номер телефона в международном формате. *PhoneField* затем обработает валидацию и представление номеров телефонов, облегчая пользователям ввод стандартизированных номеров телефонов.

Поле будет отображаться в клиентском приложении в виде поля ввода для ввода номера телефона, включая код страны.

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

Универсальное поле для кодирования различных типов данных в QR-коды.

Это поле может закодировать различные типы данных в представление QR-кода, что делает его полезным для отображения QR-кодов в пользовательском интерфейсе. Оно работает путем сериализации или десериализации данных с использованием указанного дочернего поля.

Параметры

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

Пример:

Предположим, у вас есть модель с полем *data*, и вы хотите отображать его представление в виде QR-кода в графическом интерфейсе пользователя. Вы можете использовать *QrCodeField* в своем сериализаторе:

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

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

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

В этом примере поле qr_code_data будет представлять QR-код, сгенерированный из поля данных в графическом интерфейсе пользователя. Пользователи могут взаимодействовать с этим QR-кодом, и, если их устройство поддерживает, они могут сканировать код для дополнительных действий.

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“.

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*[130], *int*[131]) – минимальный уровень
- **max_value** (*float*[132], *int*[133]) – максимальный уровень
- **step** (*float*[134], *int*[135]) – минимальный шаг между уровнями
- **front_style** (*str*[136]) – визуализация поля на фронтенде. Допустимы: [„stars“, „slider“, „fa_icon“].
- **color** (*str*[137]) – цвет элемента рейтинга (star, icon или slider) в формате css
- **fa_class** (*str*[138]) – 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*[139]) – префикс для operation_id, например, если operation_id = *history_get*, то operation_name = *history*
- **depend_field** (*str*[140]) – имя поля, от которого оно зависит, его значение будет использовано для operation_id
- **concat_field_name** (*bool*[141]) – 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.

---

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

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

Расширяет класс *VSTCharField*. Представляет обратное ForeignKey отношение в виде списка связанных экземпляров.

Это поле позволяет представить обратное ForeignKey отношение в виде списка связанных экземпляров. Для использования укажите `related_name` (related manager для обратного ForeignKey) и `fields` (список или кортеж полей из связанной модели, которые должны быть включены).

По умолчанию используется *VSTCharField* для сериализации всех значений поля и их представления на фронтенде. Вы можете указать *serializer_class* и переопределить поля по мере необходимости. Например, заголовок, описание и другие свойства полей можно установить для настройки поведения на фронтенде.

Параметры

- **related_name** (*str*[142]) – Имя related manager для обратного ForeignKey.
- **fields** (*list*[143]*[str*[144]*],* *tuple*[145]*[str*[146]*]*) – Список связанных полей модели.
- **view_type** (*str*[147]) – Определяет, как поля будут представлены на фронтенде. Должен быть либо `list`, либо `table`.
- **fields_custom_handlers_mapping** (*dict*[148]) – Отображение пользовательских обработчиков, где ключ: field_name, значение: callable_obj, принимающий параметры: instance[dict], fields_mapping[dict], model, field_name[str].
- **serializer_class** (*type*[149]) – Сериализатор для настройки типов полей. Если сериализатор не предоставлен, 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*[150]*,* *list*[151]) – 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*)

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

Примечание: Это поле предназначено для использования в графическом интерфейсе пользователя (GUI) и работает аналогично *VSTCharField* в API.

---

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

Пример:

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

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

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

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

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

Поле продолжительности времени, в секундах, специально разработанное для вычисления и отображения времени работы системы.

Это поле действует только в графическом интерфейсе и ведет себя аналогично `rest_framework.fields.IntegerField` в API.

Класс *UptimeField* преобразует время в секундах в удобочитаемое представление на фронтенде. Он интеллектуально выбирает наиболее подходящий шаблон из следующих:

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

Пример:

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

В этом примере предполагается, что сериализатор имеет поле *uptime*, представляющее продолжительность времени в секундах. *UptimeField* затем отобразит продолжительность в удобочитаемом формате на фронтенде, облегчая пользователям интерпретацию и ввод значений.

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*)

Расширяет класс *`TextareaField`* для отображения редактора https://ui.toast.com/tui-на фронтенде.

Это поле специально разработано для отображения WYSIWYG-редактора на фронтенде с использованием https://ui.toast.com/tui-editor. Сохраняет данные в формате markdown и экранирует все HTML-теги.

Параметры

escape (*bool*[152]) – Экранирование HTML-тегов для входных данных. Включено по умолчанию для предотвращения уязвимостей HTML-инъекций.

Пример:

Включите это поле в сериализаторе, чтобы отобразить WYSIWYG-редактор на фронтенде:

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

В этом примере предполагается, что сериализатор имеет поле *wysiwyg_content*, представляющее данные, которые должны быть отображены в WYSIWYG-редакторе на фронтенде. *WYSIWYGField* гарантирует, что содержимое сохраняется в формате markdown, и HTML-теги экранируются для повышения безопасности.

Предупреждение: Рекомендуется включить опцию `escape` для предотвращения потенциальных уязвимостей HTML-инъекций.

Примечание: Отображение на фронтенде достигается с использованием https://ui.toast.com/tui-editor.

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[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

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[156][typing.Tuple[157], typing.List[158], None[159]]) –

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

Обертка для ImageBaseSizeValidator, проверяющая только высоту

Параметры

- **min_height** – минимальная валидная высота изображения

[152] https://docs.python.org/3.8/library/functions.html#bool
[153] https://docs.python.org/3.8/library/typing.html#typing.Union
[154] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[155] https://docs.python.org/3.8/library/typing.html#typing.List
[156] https://docs.python.org/3.8/library/constants.html#None
[157] https://docs.python.org/3.8/library/typing.html#typing.Union
[158] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[159] https://docs.python.org/3.8/library/typing.html#typing.List
[160] https://docs.python.org/3.8/library/constants.html#None

- **max_height** – максимальная валидная высота изображения
- **extensions** (typing.Union[161][typing.Tuple[162], typing.List[163], None[164]]) –

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[165][typing.Tuple[166], typing.List[167], None[168]]) –

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[169][typing.Tuple[170], typing.List[171], None[172]]) –

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[173][typing.Tuple[174], typing.List[175], None[176]]) – 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

[161] https://docs.python.org/3.8/library/typing.html#typing.Union
[162] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[163] https://docs.python.org/3.8/library/typing.html#typing.List
[164] https://docs.python.org/3.8/library/constants.html#None
[165] https://docs.python.org/3.8/library/typing.html#typing.Union
[166] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[167] https://docs.python.org/3.8/library/typing.html#typing.List
[168] https://docs.python.org/3.8/library/constants.html#None
[169] https://docs.python.org/3.8/library/typing.html#typing.Union
[170] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[171] https://docs.python.org/3.8/library/typing.html#typing.List
[172] https://docs.python.org/3.8/library/constants.html#None
[173] https://docs.python.org/3.8/library/typing.html#typing.Union
[174] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[175] https://docs.python.org/3.8/library/typing.html#typing.List
[176] https://docs.python.org/3.8/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[177][typing.Tuple[178], typing.List[179], None[180]]) –

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[181][typing.Pattern[182]]) –

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

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

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

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*[185]) – Необходимая ширина
- **height** (*int*[186]) – 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

[177] https://docs.python.org/3.8/library/typing.html#typing.Union
[178] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[179] https://docs.python.org/3.8/library/typing.html#typing.List
[180] https://docs.python.org/3.8/library/constants.html#None
[181] https://docs.python.org/3.8/library/typing.html#typing.Optional
[182] https://docs.python.org/3.8/library/typing.html#typing.Pattern
[183] https://docs.python.org/3.8/library/typing.html#typing.Optional
[184] https://docs.python.org/3.8/library/typing.html#typing.Pattern
[185] https://docs.python.org/3.8/library/functions.html#int
[186] https://docs.python.org/3.8/library/functions.html#int

- **img** (*PIL.Image*) – объект Pillow Image
- **limits** (*dict*[187]) – 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[188].

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

Сериализатор по умолчанию с логикой для работы с объектами.

Этот сериализатор служит базовым классом для создания сериализаторов для работы с объектами, не являющимися моделями. Он расширяет класс „rest_framework.serializers.Serializer“ и включает дополнительную логику для обработки создания и обновления объектов.

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.

Пример:

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

В этом примере класс `MySerializer` расширяет `BaseSerializer` и включает дополнительное созданное поле.

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

Для любого сериализатора, показанного на фронтенде, аттрибут *_display_mode* может принимать одно из следующих значений.

DEFAULT = 'DEFAULT'

Используется, когда не задано никакого режима.

STEP = 'STEP'

Каждая группа параметров отображается на раздельных вкладках. При создании выглядит как пошаговый мастер.

[187] https://docs.python.org/3.8/library/stdtypes.html#dict
[188] https://www.django-rest-framework.org/api-guide/serializers/

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*)

Сериализатор модели по умолчанию, основанный на `rest_framework.serializers.ModelSerializer`. Узнайте больше в документации DRF[189] о том, как создавать сериализаторы модели. Этот сериализатор сопоставляет поля модели с расширенным набором полей сериализатора.

Список доступных пар, указанных в *VSTSerializer.serializer_field_mapping*. Например, чтобы установить *vstutils.api.fields.FkModelField* в сериализаторе, используйте *vstutils.models.fields.FkModelField* в модели.

Пример:

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

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

В этом примере класс `MySerializer` расширяет `VSTSerializer` и ассоциируется с моделью `MyModel`.

3.2.4 Представления

VSTUtils расширяет стандартное поведение ViewSets из Django REST Framework (DRF), предоставляя встроенные механизмы для управления представлениями моделей, которые удовлетворяют наиболее часто встречающимся паттернам разработки. Это улучшение фреймворка упрощает процесс создания надежных и масштабируемых веб-приложений, предлагая богатый набор инструментов и утилит, которые автоматизируют большую часть шаблонного кода, необходимого при разработке API. Благодаря этим расширениям, разработчики могут легко реализовать пользовательскую бизнес-логику, валидацию данных и контроль доступа с минимальными усилиями, тем самым значительно сокращая время разработки и улучшая поддерживаемость кода.

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.

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:

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

```
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*[190]) – Включает/выключает выполнение сохранения сериализатором, если передан именованный аргумент *data*. Включено по умолчанию.
- **custom_data** (*dict*[191]) – Словарь с данными, которые будут переданы в *validated_data* без валидации.
- **serializer_class** (*None*, *type*[192] *[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`[193] 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*.

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*[194]) – Имя вложенного пути. Также используется стандартное имя для связанных queryset'ов (см. *manager_name*).

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

- **arg** (*str*[195]) – 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*[196]) – Флаг, разрешающий добавление существующих экземпляров.
- **manager_name** (*str*[197], *Callable*[198]) – Имя атрибута объекта модели, который содержит вложенный queryset.
- **methods** (*list*[199]) – Список разрешенных методов для endpoint'ов вложенных view.
- **subs** (*list*[200], *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

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

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

Декоратор, оборачивающий метод объекта в subaction viewset'a.

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.

ETag (Entity Tag) — это механизм, определенный протоколом HTTP для проверки кэша веба и эффективного управления версиями ресурсов. Он представляет собой уникальный идентификатор содержимого ресурса на определенный момент времени, позволяя клиенту и серверу определить, изменился ли ресурс без необходимости загружать весь контент. Этот механизм значительно сокращает использование полосы пропускания и улучшает производительность веба за счет использования условных запросов. Серверы отправляют ETags в ответах клиентам, которые могут кэшировать эти теги вместе с ресурсами. При последующих запросах клиенты отправляют кэшированный ETag обратно на сервер в заголовке If-None-Match. Если ресурс не изменился (ETag совпадает), сервер отвечает статусом 304 Not Modified, указывая, что кэшированная версия клиента актуальна.

Помимо GET-запросов, ETags также могут быть полезны в других методах HTTP, таких как PUT или DELETE, для обеспечения согласованности и предотвращения непреднамеренных перезаписей или удалений, известных как «избежание столкновений в воздухе». Включая ETag в заголовок If-Match не-GET запросов, клиенты сигнализируют, что операция должна продолжаться только в том случае, если текущее состояние ресурса соответствует указанному ETag, тем самым защищая от одновременных изменений разными клиентами. Это применение ETags повышает надежность и целостность веб-приложений, гарантируя выполнение операций с корректной версией ресурса.

Вот основные функции, предоставляемые для работы с механизмом ETag:

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

Миксин, предназначенный для улучшения кэширования ответов на GET-запросы в представлениях Django REST framework, используя стандартный механизм кэширования HTTP. Он возвращает статус 304 Not Modified для чтения запросов, таких как GET или HEAD, когда ETag (Entity Tag) в запросе клиента совпадает с текущим состоянием ресурса, и статус 412 Precondition Failed для записывающих запросов, когда условие не выполняется. Этот подход снижает ненужный сетевой трафик и время загрузки для неизменных ресурсов, а также обеспечивает согласованность данных для операций записи.

Миксин использует функции *get_etag_value()* и *check_request_etag()* в контексте *GenericViewSet* для динамического создания и проверки ETag для состояний ресурсов. Сравнивая ETag, он определяет, изменилось ли содержимое с момента последнего запроса, позволяя клиентам

повторно использовать кэшированные ответы, когда это применимо, и предотвращая одновременные операции записи от перезаписи друг друга без подтверждения обновленного состояния.

Предупреждение: Этот миксин разработан для работы с моделями, наследующими от *vstutils.models.BModel*. Использование с другими моделями может не обеспечить предполагаемое поведение кэширования и может привести к некорректному поведению приложения.

Примечание: Для эффективного использования убедитесь, что классы моделей совместимы с генерацией и проверкой ETag, реализовав метод *get_etag_value()* для пользовательского вычисления ETag. Кроме того, *GenericViewSet*, использующий этот миксин, должен правильно обрабатывать заголовки ETag в запросах клиентов для использования кэширования HTTP.

Дополнительной особенностью `CachableHeadMixin` является его автоматическое включение в сгенерированное представление из *vstutils.models.BModel*, если класс модели имеет атрибут класса *_cache_responses*, установленный в `True`. Это позволяет автоматизировать возможности кэширования для моделей, указывающих на готовность к кэшированию на основе HTTP, оптимизируя процесс сокращения времени ответа и снижения нагрузки на сервер для часто доступных ресурсов.

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

Пользовательское перечисление, определяющее потенциальные зависимости для генерации ETag. Оно включает:

LANG = 4

Указывает на зависимость от языковых предпочтений пользователя.

SESSION = 2

Указывает на зависимость от сессии пользователя.

USER = 1

Указывает на зависимость от идентичности пользователя.

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

Функция играет ключевую роль в контексте механизма ETag, предоставляя гибкий способ проверки ETags на стороне клиента по сравнению с версией на стороне сервера как для проверки кэша, так и для обеспечения согласованности данных в веб-приложениях. Она поддерживает условную обработку HTTP-запросов на основе совпадения или несовпадения значений ETag, адаптируясь к различным сценариям, таким как проверки свежести кэша и предотвращение одновременных изменений.

Параметры

- **request** (`rest_framework.request.Request`) – Объект HTTP-запроса, содержащий заголовки клиента, из которых извлекается ETag для сравнения.
- **etag_value** (`str`[201]) – ETag, сгенерированный сервером, представляет текущее состояние ресурса. Этот уникальный идентификатор пересчитывается при каждом изменении содержимого ресурса.
- **header_name** (`str`[202]) – Указывает HTTP-заголовок для поиска ETag клиента. По умолчанию используется «If-None-Match», обычно используемый в GET-запросах для проверки кэша. Для операций, требующих подтверждения того, что клиент действует с последней версией ресурса (например, PUT или DELETE), вместо этого следует использовать «If-Match».

- **operation_handler** – Функция для сравнения ETags. По умолчанию установлено `str.__eq__`, которое проверяет точное совпадение ETags клиента и сервера, подходящее для проверки кэшей с `If-None-Match`. Для обработки сценариев `If-Match`, когда операция должна продолжаться только в случае, если ETags не совпадают, указывая на то, что ресурс был изменен, можно использовать `str.__ne__` (не равно) в качестве обработчика операции. Эта гибкость позволяет точно контролировать, как и когда клиентам разрешено читать или записывать ресурсы на основе их версии.

Результат

Возвращает кортеж, содержащий ETag сервера и булев флаг. Флаг равен `True`, если условие обработчика операции между ETag сервера и клиента выполнено, указывая на то, что запрос должен продолжаться на основе логики сопоставления, определенной обработчиком операции; в противном случае возвращает `False`.

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

Функция get_etag_value предназначена для вычисления уникального значения ETag на основе состояния модели, параметров запроса и дополнительных зависимостей, таких как язык пользователя, сессия и идентичность пользователя. Эта функция поддерживает как отдельные модели, так и коллекции моделей.

Параметры

- **view** (*GenericViewSet*) – Экземпляр *GenericViewSet*, отвечающий за операции просмотра.
- **model_class** (`django.db.models.Model`[203], `list`[204], `tuple`[205], or `set`[206]) – Класс модели, для которой генерируется значение ETag. Этот параметр может быть одним классом модели или коллекцией классов моделей (`list`[207], `tuple`[208] или `set`[209]). Каждый класс модели может необязательно реализовать метод класса с именем `get_etag_value`.
- **request** (`rest_framework.request.Request`) – Объект запроса из Django REST framework, содержащий всю информацию HTTP-запроса.
- **pk** – Первичный ключ экземпляра модели, для которого рассчитывается ETag. Это может быть конкретное значение (int или str) для использования в одной модели или словарь, сопоставляющий классы моделей с их соответствующими значениями первичных ключей для коллекций моделей.

Результат

Вычисленное значение ETag в виде шестнадцатеричной строки.

Тип результата

`str`[210]

Функция работает по-разному в зависимости от предоставленного типа model_class:

- **Коллекция моделей**: Когда model_class является коллекцией, функция вычисляет ETag, объединяя значения ETag отдельных моделей в коллекции, используя рекурсивный вызов для каждой модели. Значение ETag для каждой модели кодируется и хешируется с использованием алгоритма Blake2s.
- **Модель с методом ``get_etag_value``**: Если класс модели имеет метод get_etag_value, функция вызывает этот метод, чтобы получить базовое значение ETag. Затем она добавляет информацию о языке, идентификаторе пользователя и ключе сессии, если они помечены как зависимости в атрибуте `_cache_response_dependencies` модели. Это базовое значение ETag может быть дополнительно обработано, включая полную строку версии приложения и хешировано, если включена информация о пользователе или сессии.

[201] https://docs.python.org/3.8/library/stdtypes.html#str
[202] https://docs.python.org/3.8/library/stdtypes.html#str

- **Модель без метода ``get_etag_value``**: Для моделей без пользовательского метода get_etag_value функция генерирует ETag на основе имени класса модели и хеша полной строки версии приложения.

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*, *require_confirmation=False*, ***kwargs*)

Базовый класс действий. Обладает минимально необходимой функциональностью для создания действия 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 = ...
```

(continues on next page)

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

(продолжение с предыдущей страницы)

```
    ...

    @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.
- **require_confirmation** – Если истина, то в интерфейсе пользователь должен будет подтвердить действие перед выполнением.
- **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
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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.
↪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[211]) – Имя поля первичного ключа в связанной модели. По умолчанию „id“.

- **related_name** (str[212]) – Имя charfield-поля в связанной модели. По умолчанию „name“.
- **pk_handler** (typing.Callable[213]) – 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*[214]) – queryset модели для фильтрации.
- **field** (*str*[215]) – имя поля в FilterSet'е. Также поддерживает суффикс *__not*.
- **value** (*str*[216]) – список искомых значений, разделенных запятыми.

Результат

отфильтрованный queryset.

Тип результата

django.db.models.query.QuerySet[217]

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*[218]) – queryset модели для фильтрации.
- **field** (*str*[219]) – имя поля в FilterSet'е. Также поддерживает суффикс *__not*.
- **value** (*str*[220]) – часть названия для поиска.

Результат

отфильтрованный queryset.

Тип результата

django.db.models.query.QuerySet[221]

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

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*[222]) – Код состояния HTTP.
- **timings** (*int*[223], *None*) – Тайминги ответов.

Параметры

timings – Тайминги ответов.

3.2.8 Middleware

По умолчанию Django предполагает[224], 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
```

(continues on next page)

[222] https://docs.python.org/3.8/library/functions.html#int
[223] https://docs.python.org/3.8/library/functions.html#int
[224] https://docs.djangoproject.com/en/3.2/topics/http/middleware/#writing-your-own-middleware

(продолжение с предыдущей страницы)

```
    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.

> **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*[225]) – Объект HTTP-запроса, созданный из клиентского запроса.

Тип результата

django.http.HttpResponse[226]

handler(*request*, *response*)

Handler ответа. Метод для обработки ответов.

Параметры

- **request** (*django.http.HttpRequest*[227]) – Объект HTTP-запроса.
- **response** (*django.http.HttpResponse*[228]) – Объект HTTP-ответа, который будет отправлен клиенту.

Результат

Обработанный объект ответа.

Тип результата

django.http.HttpResponse[229]

request_handler(*request*)

Handler запроса. Вызывается перед обработкой запроса view.

Параметры
request (*django.http.HttpRequest*[230]) – Объект HTTP-запроса, созданный из клиентского запроса.

Результат
Обработанный объект запроса.

Тип результата
django.http.HttpRequest[231]

3.2.9 Filter Backend'ы

Filter Backend'ы[232] 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*.

[225] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[226] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[227] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[228] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[229] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[230] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[231] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpRequest
[232] https://www.django-rest-framework.org/api-guide/filtering/#djangofilterbackend

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[233].

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.

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 [
```

(continues on next page)

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

(продолжение с предыдущей страницы)

```
                {
                    "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[234]. 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“]

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

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()*

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[235]

operate(*operation_data*, *context*)

Метод, используемый для обработки одной операции и возвращающий ее результат

Параметры

- **operation_data** (typing.Dict[236]) –
- **context** (typing.Dict[237]) –

Тип результата
typing.Tuple[238][typing.Dict[239], typing.SupportsFloat[240]]

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

[235] https://docs.python.org/3.8/library/stdtypes.html#dict
[236] https://docs.python.org/3.8/library/typing.html#typing.Dict
[237] https://docs.python.org/3.8/library/typing.html#typing.Dict
[238] https://docs.python.org/3.8/library/typing.html#typing.Tuple
[239] https://docs.python.org/3.8/library/typing.html#typing.Dict
[240] https://docs.python.org/3.8/library/typing.html#typing.SupportsFloat

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)
- `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[241] (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

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

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:

```
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
        )
        # 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)
        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`[242].

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`[243]) –
- **second** (`typing.Dict`[244]) –
- **msg** (`str`[245]) –

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

Вызывает `len()`[246] через `iterable` и проверяет равенство с `count`.

Параметры

- **iterable** (`typing.Sized`[247]) – любой итерируемый объект, который может быть отправлен в `len()`[248].
- **count** (`int`[249]) – ожидаемый результат.
- **msg** (`typing.Any`[250]) – сообщение об ошибке

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

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

Параметры

- **resp** (*`django.http.HttpResponse`*[251]) – объект ответа
- **code** (*`int`*[252]) – ожидаемый код

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

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

Параметры

- **data** (`typing.Union`[253][`typing.List`[254][`typing.Dict`[255][`str`[256], `typing.Any`[257]]], `str`[258], `bytes`[259], `bytearray`[260]) – данные запроса
- **code** (`int`[261]) – http-статус для проверки
- **kwargs** – именованные аргументы для *`get_result()`*

Тип результата

`typing.Union`[262][`typing.List`[263][`typing.Dict`[264][`str`[265], `typing.Any`[266]]], `str`[267], `bytes`[268], `bytearray`[269], `typing.Dict`[270], `typing.Sequence`[271][`typing.Union`[272][`typing.List`[273][`typing.Dict`[274][`str`[275], `typing.Any`[276]]], `str`[277], `bytes`[278], `bytearray`[279]]]]

Результат

bulk-ответ

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

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

Параметры

- **data** (typing.Union[280][typing.List[281][typing.Dict[282][str[283], typing.Any[284]]], str[285], bytes[286], bytearray[287]) – данные запроса
- **code** (int[288]) – http-статус для проверки
- **kwargs** – именованные аргументы для *get_result()*

Тип результата
typing.Union[289][typing.List[290][typing.Dict[291][str[292], typing.Any[293]]], str[294], bytes[295], bytearray[296], typing.Dict[297], typing.Sequence[298][typing.Union[299][typing.List[300][typing.Dict[301][str[302], typing.Any[303]]], str[304], bytes[305], bytearray[306]]]

Результат
bulk-ответ

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

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

Параметры

- **data** (*dict*[307]) – 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[308][typing.List[309][typing.Dict[310][str[311], typing.Any[312]]], str[313], bytes[314], bytearray[315]) – данные запроса
- **method** (str[316]) – метод http-запроса
- **code** (int[317]) – http-статус для проверки
- **query** – словарь с данными query (работает только с *get*)

Тип результата
typing.Union[318][typing.List[319][typing.Dict[320][str[321], typing.Any[322]]], str[323], bytes[324], bytearray[325], typing.Dict[326], typing.Sequence[327][typing.Union[328][typing.List[329][typing.Dict[330][str[331], typing.Any[332]]], str[333], bytes[334], bytearray[335]]]

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*[336], *django.db.models.Model*[337]) – строка, содержащая имя модели (если атрибут `model` установлен в класс тест-кейса), импорт модуля, `app.ModelName` или `django.db.models.Model`[338].
- **kwargs** – именованные аргументы для `django.db.models.query.QuerySet.filter()`[339].

Результат
количество объектов в базе данных.

Тип результата
int[340]

get_model_class(*model*)

Получение класса модели по строке или получение аргумента модели.

Параметры
model (*str*[341], *django.db.models.Model*[342]) – строка, содержащая имя модели (если атрибут `model` установлен в класс тест-кейса), импорт модуля, `app.ModelName` или `django.db.models.Model`[343].

Результат
Класс модели.

Тип результата
django.db.models.Model[344]

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

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

Параметры

- **model** (*str*[345], *django.db.models.Model*[346]) – строка, содержащая имя модели (если атрибут `model` установлен в класс тест-кейса), импорт модуля, `app.ModelName` или `django.db.models.Model`[347].
- **kwargs** – именованные аргументы для `django.db.models.query.QuerySet.filter()`[348].

Тип результата
django.db.models.query.QuerySet[349]

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[350]) – 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[351][typing.List[352][typing.Dict[353][str[354], typing.Any[355]]], str[356], bytes[357], bytearray[358], typing.Dict[359], typing.Sequence[360][typing.Union[361][typing.List[362][typing.Dict[363][str[364], typing.Any[365]]], str[366], bytes[367], bytearray[368]]]

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[369]

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()[370].

Тип результата
typing.ContextManager[371][unittest.mock.Mock[372]]

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()[373].

Параметры

- **model** (django.db.models.base.Model) –
- **field_name** (str[374]) –
- **value** (typing.Any[375]) –

Тип результата
typing.ContextManager[376][unittest.mock.Mock[377]]

random_name()

Простая функция, возвращающая строку uuid1.

Тип результата
str[378]

std_codes: typing.Dict[379][str[380], int[381]] = {'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*[382]) – новый объект пользователя, от которого будет выполнение.

[242] https://docs.djangoproject.com/en/4.2/topics/testing/tools/#django.test.TestCase
[243] https://docs.python.org/3.8/library/typing.html#typing.Dict
[244] https://docs.python.org/3.8/library/typing.html#typing.Dict
[245] https://docs.python.org/3.8/library/stdtypes.html#str
[246] https://docs.python.org/3.8/library/functions.html#len
[247] https://docs.python.org/3.8/library/typing.html#typing.Sized
[248] https://docs.python.org/3.8/library/functions.html#len
[249] https://docs.python.org/3.8/library/functions.html#int
[250] https://docs.python.org/3.8/library/typing.html#typing.Any
[251] https://docs.djangoproject.com/en/4.2/ref/request-response/#django.http.HttpResponse
[252] https://docs.python.org/3.8/library/functions.html#int
[253] https://docs.python.org/3.8/library/typing.html#typing.Union
[254] https://docs.python.org/3.8/library/typing.html#typing.List
[255] https://docs.python.org/3.8/library/typing.html#typing.Dict
[256] https://docs.python.org/3.8/library/stdtypes.html#str
[257] https://docs.python.org/3.8/library/typing.html#typing.Any
[258] https://docs.python.org/3.8/library/stdtypes.html#str
[259] https://docs.python.org/3.8/library/stdtypes.html#bytes
[260] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[261] https://docs.python.org/3.8/library/functions.html#int
[262] https://docs.python.org/3.8/library/typing.html#typing.Union
[263] https://docs.python.org/3.8/library/typing.html#typing.List
[264] https://docs.python.org/3.8/library/typing.html#typing.Dict
[265] https://docs.python.org/3.8/library/stdtypes.html#str
[266] https://docs.python.org/3.8/library/typing.html#typing.Any
[267] https://docs.python.org/3.8/library/stdtypes.html#str
[268] https://docs.python.org/3.8/library/stdtypes.html#bytes
[269] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[270] https://docs.python.org/3.8/library/typing.html#typing.Dict
[271] https://docs.python.org/3.8/library/typing.html#typing.Sequence
[272] https://docs.python.org/3.8/library/typing.html#typing.Union
[273] https://docs.python.org/3.8/library/typing.html#typing.List
[274] https://docs.python.org/3.8/library/typing.html#typing.Dict
[275] https://docs.python.org/3.8/library/stdtypes.html#str
[276] https://docs.python.org/3.8/library/typing.html#typing.Any
[277] https://docs.python.org/3.8/library/stdtypes.html#str
[278] https://docs.python.org/3.8/library/stdtypes.html#bytes
[279] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[280] https://docs.python.org/3.8/library/typing.html#typing.Union
[281] https://docs.python.org/3.8/library/typing.html#typing.List
[282] https://docs.python.org/3.8/library/typing.html#typing.Dict
[283] https://docs.python.org/3.8/library/stdtypes.html#str
[284] https://docs.python.org/3.8/library/typing.html#typing.Any
[285] https://docs.python.org/3.8/library/stdtypes.html#str
[286] https://docs.python.org/3.8/library/stdtypes.html#bytes
[287] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[288] https://docs.python.org/3.8/library/functions.html#int
[289] https://docs.python.org/3.8/library/typing.html#typing.Union
[290] https://docs.python.org/3.8/library/typing.html#typing.List
[291] https://docs.python.org/3.8/library/typing.html#typing.Dict
[292] https://docs.python.org/3.8/library/stdtypes.html#str
[293] https://docs.python.org/3.8/library/typing.html#typing.Any
[294] https://docs.python.org/3.8/library/stdtypes.html#str
[295] https://docs.python.org/3.8/library/stdtypes.html#bytes
[296] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[297] https://docs.python.org/3.8/library/typing.html#typing.Dict
[298] https://docs.python.org/3.8/library/typing.html#typing.Sequence
[299] https://docs.python.org/3.8/library/typing.html#typing.Union
[300] https://docs.python.org/3.8/library/typing.html#typing.List
[301] https://docs.python.org/3.8/library/typing.html#typing.Dict
[302] https://docs.python.org/3.8/library/stdtypes.html#str
[303] https://docs.python.org/3.8/library/typing.html#typing.Any
[304] https://docs.python.org/3.8/library/stdtypes.html#str
[305] https://docs.python.org/3.8/library/stdtypes.html#bytes
[306] https://docs.python.org/3.8/library/stdtypes.html#bytearray
[307] https://docs.python.org/3.8/library/stdtypes.html#dict
[308] https://docs.python.org/3.8/library/typing.html#typing.Union
[309] https://docs.python.org/3.8/library/typing.html#typing.List
[310] https://docs.python.org/3.8/library/typing.html#typing.Dict
[311] https://docs.python.org/3.8/library/stdtypes.html#str
[312] https://docs.python.org/3.8/library/typing.html#typing.Any
[313] https://docs.python.org/3.8/library/stdtypes.html#str
[314] https://docs.python.org/3.8/library/stdtypes.html#bytes
[315] https://docs.python.org/3.8/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*[383]) – название параметра
- **default** (*object*[384]) – 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

[383] https://docs.python.org/3.8/library/stdtypes.html#str
[384] https://docs.python.org/3.8/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[385]) –

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*[386]) – – 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*[387], *str*[388]) – – уникальный id блокировки.
- **payload** – – дополнительная информация о блокировке. Должна быть значением, равным True при приведении к булевому типу.
- **repeat** (*int*[389]) – – время ожидания lock.release. По умолчанию 1 секунда.
- **err_msg** (*str*[390]) – – сообщение для ошибки AcquireLockException.

Примечание:

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

Пример:

[385] https://docs.python.org/3.8/library/stdtypes.html#str
[386] https://docs.python.org/3.8/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*[391]) – – словарь объектов, например {<name>: <backend_class>}
- **keys** (*list*[392]) – – имена поддерживаемых бэкендов
- **values** (*list*[393]) – – 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*[394]) – – объект модели

[387] https://docs.python.org/3.8/library/functions.html#int
[388] https://docs.python.org/3.8/library/stdtypes.html#str
[389] https://docs.python.org/3.8/library/functions.html#int
[390] https://docs.python.org/3.8/library/stdtypes.html#str

Результат
объект бэкенда

Тип результата
object[395]

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*[396]) – Имя для бэкенда, наподобие ключа в словаре.

backend(*name*)

Получить класс бэкенда

Параметры
name (*str*[397]) – – имя типа бэкенда

Результат
класс бэкенда

Тип результата
type[398],types.ModuleType[399],object[400]

[391] https://docs.python.org/3.8/library/stdtypes.html#dict
[392] https://docs.python.org/3.8/library/stdtypes.html#list
[393] https://docs.python.org/3.8/library/stdtypes.html#list
[394] https://docs.djangoproject.com/en/4.2/ref/models/instances/#django.db.models.Model
[395] https://docs.python.org/3.8/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*[401]) – регулярное выражение url'а
- **argv** – переопределенные аргументы
- **kwargs** – переопределенные kwarg'и

[396] https://docs.python.org/3.8/library/stdtypes.html#str
[397] https://docs.python.org/3.8/library/stdtypes.html#str
[398] https://docs.python.org/3.8/library/functions.html#type
[399] https://docs.python.org/3.8/library/types.html#types.ModuleType
[400] https://docs.python.org/3.8/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[402]) –

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.

[401] https://docs.python.org/3.8/library/stdtypes.html#str
[402] https://docs.python.org/3.8/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.

Предупреждение: Эта функция олдскульная и будет объявлена устаревшей в будущих версиях. Используйте встроенный вызов метода **:method:`vstutils.models.BModel.get_view_class`**.

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*[403]) – – секретный ключ для кодирования
- **enc** (*str*[404]) – – закодированная строка для декодирования

Результат

– декодированная строка

Тип результата

str[405]

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

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

[403] https://docs.python.org/3.8/library/stdtypes.html#str
[404] https://docs.python.org/3.8/library/stdtypes.html#str
[405] https://docs.python.org/3.8/library/stdtypes.html#str

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

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

Закодировать строку шифром Виженера.

Параметры

- **key** (*str*[406]) – – секретный ключ для кодирования
- **clear** (*str*[407]) – – чистое значение для кодирования

Результат
– закодированная строка

Тип результата
str[408]

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

Рендеринг строки из шаблона.

Параметры

- **name** (*str*[409]) – – полное название шаблона
- **data** (*dict*[410]) – – словарь переменных для рендеринга
- **trans** (*str*[411]) – – перевод для рендера. По умолчанию „en“.

Результат
– отрендеренная строка

Тип результата
str[412]

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*

[406] https://docs.python.org/3.8/library/stdtypes.html#str
[407] https://docs.python.org/3.8/library/stdtypes.html#str
[408] https://docs.python.org/3.8/library/stdtypes.html#str
[409] https://docs.python.org/3.8/library/stdtypes.html#str
[410] https://docs.python.org/3.8/library/stdtypes.html#dict
[411] https://docs.python.org/3.8/library/stdtypes.html#str
[412] https://docs.python.org/3.8/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()`[413], 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.

[413] 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*[414]) – – строка для записи во временный файл.
- **mode** (*str*[415]) – – режим открытия файла. По умолчанию *w*.
- **bufsize** (*int*[416]) – – 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*[417]) – – 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()`[418] для получения кода языка и пытается получить перевод из списка доступных.

Параметры

text – Текстовое сообщение, которое должно быть переведено.

3.7 Интеграция Web Push-уведомлений

Web-уведомления - это эффективный способ взаимодействия с пользователями с помощью реального времени. Чтобы интегрировать web-уведомления в ваш проект VSTUtils, выполните следующие шаги:

1. **Конфигурация**: Во-первых, включите модуль `vstutils.webpush` в разделе `INSTALLED_APPS` вашего файла `settings.py`. Это позволяет использовать функциональность web-уведомлений, предоставляемую VSTUtils. Кроме того, настройте необходимые параметры, как описано в разделе настроек web-уведомлений (см. *здесь* для подробностей).

2. **Создание уведомлений**: Чтобы создать web-уведомление, вам нужно определить класс, который наследуется от `vstutils.webpush.BaseWebPush` или `vstutils.webpush.BaseWebPushNotification`. VSTUtils автоматически обнаруживает и использует классы web-уведомлений, определенные в модуле `webpushes` всех `INSTALLED_APPS`. Ниже приведен пример, иллюстрирующий, как реализовать пользовательские классы web-уведомлений:

```
from vstutils.api.models import Language
from vstutils.webpush.base import BaseWebPush, BaseWebPushNotification
from vstutils.webpush.models import WebPushDeviceSubscription,␣
↪WebPushNotificationSubscription


class TestWebPush(BaseWebPush):
    """
    Webpush that is sent to all subscribed users
    """

    def get_subscriptions(self):
        return WebPushDeviceSubscription.objects.filter(
            user_id__in=WebPushNotificationSubscription.objects.filter(
                type=self.get_key(),
                enabled=True,
            ).values('user_id'),
        )

```

(continues on next page)

[414] https://docs.python.org/3.8/library/stdtypes.html#str
[415] https://docs.python.org/3.8/library/stdtypes.html#str
[416] https://docs.python.org/3.8/library/functions.html#int
[417] https://docs.python.org/3.8/library/stdtypes.html#str
[418] https://docs.djangoproject.com/en/4.2/ref/utils/#django.utils.translation.get_language

(продолжение с предыдущей страницы)

```
    def get_payload(self, lang: Language):
        return {"some": "data", "lang": lang.code}


class TestNotification(BaseWebPushNotification):
    """
    Webpush notification that is sent only to selected users
    """

    def __init__(self, name: str, user_id: int):
        self.name = name
        self.user_id = user_id
        self.message = f"Hello {self.name}"

    def get_users_ids(self):
        return (self.user_id,)

    def get_notification(self, lang: Language):
        return {
            "title": self.message,
            "options": {
                "body": "Test notification body",
                "data": {"url": "/"},
            },
        }


class StaffOnlyNotification(BaseWebPushNotification):
    """
    Webpush notification that only staff user can subscribe to.
    """

    @staticmethod
    def is_available(user):
        return user.is_staff
```

Этот пример содержит три класса:

- *TestWebPush*: Отправляет уведомления всем подписанным пользователям.
- *TestNotification*: Направляет уведомления конкретным пользователям.
- *StaffOnlyNotification*: Ограничивает уведомления только для сотрудников. Иногда вы можете хотеть разрешить подписку на конкретные уведомления только некоторым пользователям.

3. **Отправка уведомлений**: Чтобы отправить web-уведомление, вызовите метод `send` или `send_in_task` на экземпляре вашего класса web-уведомления. Например, чтобы отправить уведомление с использованием *TestNotification*, вы можете сделать следующее:

```
from test_proj.webpushes import TestNotification

# Sending a notification immediately (synchronously)
TestNotification(name='Some user', user_id=1).send()

# Sending a notification as a background task (asynchronously)
TestNotification.send_in_task(name='Some user', user_id=1)
```

Предупреждение: Асинхронная отправка web-уведомлений (с использованием методов, таких как `send_in_task`) требует настроенной конфигурации Celery в вашем проекте, поскольку она полагается на задачи Celery «под капотом». Убедитесь, что Celery правильно настроен и работает, чтобы использовать асинхронную отправку уведомлений.

Следуя этим шагам, вы быстро сможете интегрировать и использовать web-уведомления в проектах с VSTUtils.

4

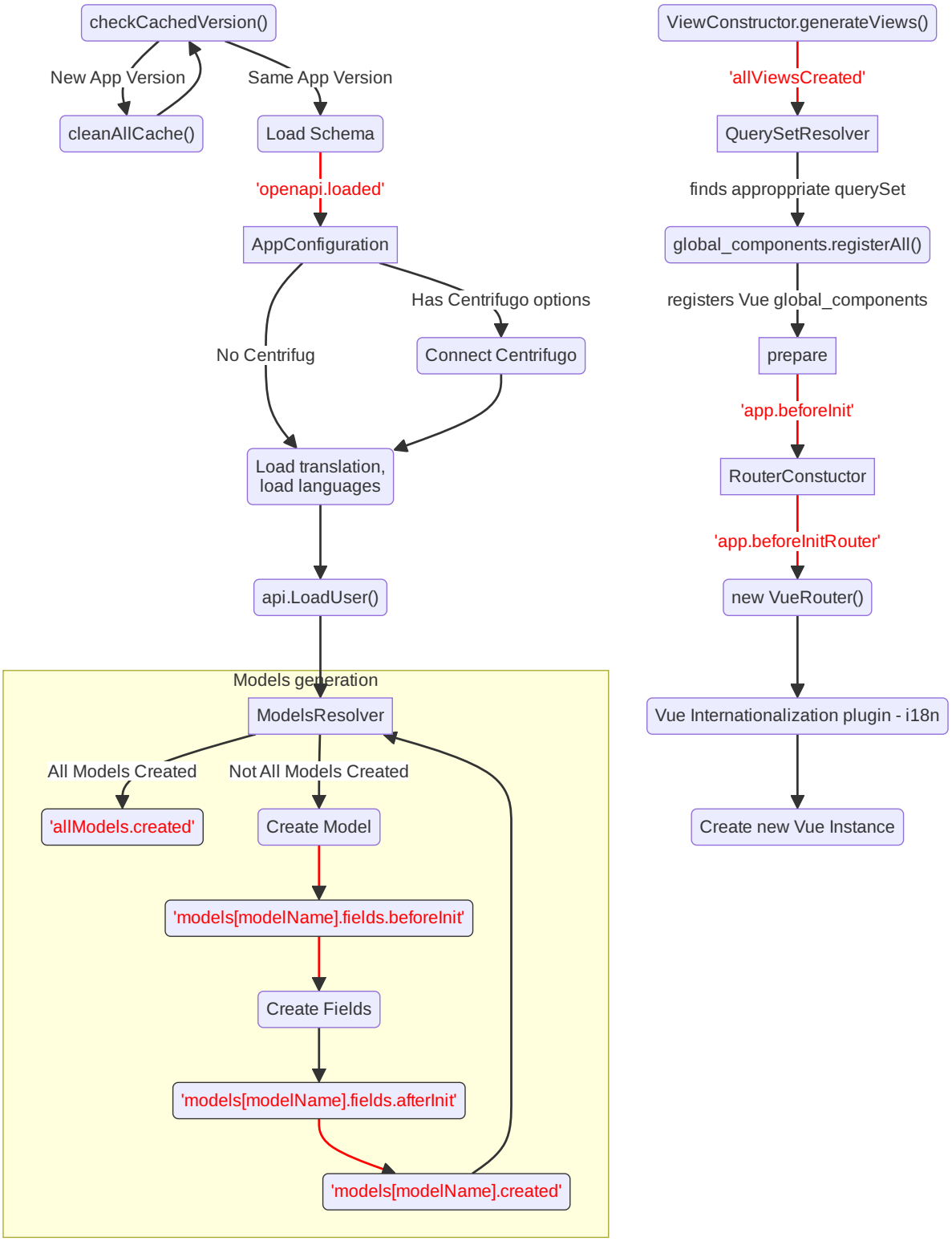
Быстрый старт фронтенда

Фреймворк VST utils использует экосистему Vue для отображения фронтенда. Руководство по быстрому старту проведет вас через наиболее важные шаги по настройке функций фронтенда. Установка приложения и настройка описаны в - *разделе Backend* этой документации.

Есть несколько этапов в приложении VST utils:

1. Перед запуском приложения:
    - *checkCacheVersions()* проверяет, была ли изменена версия приложения с последнего посещения и очищает все кэшированные данные, если это так;
    - загрузка схемы OpenAPI с бэкенда. Генерирует сигнал „openapi.loaded“;
    - загрузка всех статических файлов из *SPA_STATIC* в setting.py;
    - устанавливает *AppConfiguration* из схемы OpenAPI;
2. Приложение запущено:
    - если в settings.py есть centrifugoClient, подключается к нему. Дополнительные сведения о конфигурации centrifugo можно найти в разделе «*Настройки клиента Centrifugo*»;
    - загрузка списка доступных языков и переводов;
    - *api.loadUser()* возвращает данные пользователя;
    - *ModelsResolver* создает модели из схемы, генерирует сигнал *models[${modelName}].created* для каждой созданной модели и *allModels.created*, когда все модели созданы;
    - *ViewConstructor.generateViews()* инициализирует *View* fieldClasses и modelClasses;
    - *QuerySetsResolver* находит соответствующий queryset по имени модели и пути представления;
    - *global_components.registerAll()* регистрирует Vue *global_components*;
    - *prepare()* генерирует сигнал *app.beforeInit* с { app: this };
    - инициализация модели с *LocalSettings*. Узнайте больше об этом в разделе *Локальные настройки*;
    - создание routerConstructor из *this.views*, генерация сигнала „app.beforeInitRouter“ с { routerConstructor } и получение нового VueRouter({this.routes});
    - инициализация приложения *Vue()* из schema.info, хранилища pinia и генерация сигнала „app.afterInit“ с {app: this};
3. Приложение смонтировано.

Есть схема, представляющая процесс инициализации приложения (названия сигналов красным шрифтом):



4.1 Настройка поля

Чтобы добавить собственный скрипт в проект, укажите его имя в settings.py

```
SPA_STATIC += [
    {'priority': 101, 'type': 'js', 'name': 'main.js', 'source': 'project_lib'},
]
```

и поместите скрипт (*main.js*) в каталог *{appName}/static/*.

1. В *main.js* создайте новое поле, расширив его от BaseField (или любого другого соответствующего поля)

   Например, создадим поле, которое отображает элемент HTML h1 с текстом „Привет, мир!“

```
class CustomField extends spa.fields.base.BaseField {
    static get mixins() {
        return super.mixins.concat({
            render(createElement) {
                return createElement('h1', {}, 'Hello World!');
            },
        });
    }
}
```

Или отобразите имя человека с некоторым префиксом

```
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. Зарегистрируйте это поле в *app.fieldsResolver*, чтобы предоставить соответствующий формат и тип поля для нового поля

```
const customFieldFormat = 'customField';
app.fieldsResolver.registerField('string', customFieldFormat, CustomField);
```

3. Слушайте подходящий сигнал *models[ModelWithFieldToChange].fields.beforeInit* для изменения формата поля

```
spa.signals.connect(`models[ModelWithFieldToChange].fields.beforeInit`, (fields) => {
    fields.fieldToChange.format = customFieldFormat;
});
```

Список моделей и их полей доступен во время выполнения в консоли по адресу *app.modelsClasses*

Чтобы изменить поведение поля, создайте новый класс поля с необходимой логикой. Допустим, вам нужно отправить API количество миллисекунд, но пользователь хочет вводить количество секунд. Решением будет переопределить методы *toInner* и *toRepresent* поля.

```
class MilliSecondsField extends spa.fields.numbers.integer.IntegerField {
  toInner(data) {
    return super.toInner(data) * 1000;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  }
  toRepresent(data) {
    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;
});
```

Теперь у вас есть поле, которое отображает секунды, но сохраняет/получает данные в миллисекундах в виде детализированного представления модели AllFieldsModel.

Примечание: Если вам нужно показать какое-то предупреждение или ошибку в консоли разработчика, вы можете использовать методы *warn* и *error* поля. Вы можете передать сообщение, и оно будет выведено с типом поля, именем модели и именем поля.

4.2 Изменение пути к полю FkField

Иногда вам может потребоваться запросить другой набор объектов для FkField. Например, чтобы выбрать только известных авторов, создайте конечную точку *famous_author* на бэкенде и установите путь запроса FkField в *famous_author*. Слушайте сигнал *app.beforeInit*.

```
spa.signals.connect('app.beforeInit', ({ app }) => {
  app.modelsResolver.get('OnePost').fields.get('author').querysets.get('/post/new/
↪')[0].url = '/famous_author/'
});
```

Теперь, когда мы создаем новый пост на конечной точке */post/*, Author FkField выполняет GET-запрос к */famous_author/* вместо */author/*. Это полезно для получения другого набора авторов (которые могли быть ранее отфильтрованы на бэкенде).

4.3 Стилизация CSS

1. Как и скрипты, файлы CSS можно добавить в SPA_STATIC в setting.py

```
SPA_STATIC += [

    {'priority': 101, 'type': 'css', 'name': 'style.css', 'source': 'project_lib'},
]
```

Давайте проанализируем страницу и найдем класс CSS для нашего пользовательского поля. Это *column-format-customField* и создается с использованием шаблона *column-format-{Field.format}*.

2. Используйте обычные стили CSS для изменения внешнего вида поля.

```
.column-format-customField:hover {
    background-color: orangered;
    color: white;
}
```

Другие элементы страницы также доступны для стилизации: например, чтобы скрыть определенный столбец, установите соответствующее поле в значение none.

```
.column-format-customField {
    display: none;
}
```

4.4 Показать столбец первичных ключей в списке

Каждый столбец первичного ключа имеет класс CSS *pk-column* и по умолчанию скрыт (с использованием *display: none;*).

Например, этот стиль покажет столбец первичных ключей во всех представлениях списка модели *Order*.

```
.list-Order .pk-column {
    display: table-cell;
}
```

4.5 Настройка представления

Слушайте сигнал *«allViews.created»* и добавьте новый пользовательский миксин в представление.

В следующем фрагменте кода показано отображение нового представления вместо представления по умолчанию.

```
spa.signals.once('allViews.created', ({ views }) => {
    const AuthorListView = views.get('/author/');
    AuthorListView.mixins.push({
        render(h) {
            return h('h1', {}, `Custom view`);
        },
    });
});
```

Узнайте больше о функции *render()* Vue в документации Vue[419].

Также можно настроить представление, переопределив вычисляемые свойства и методы по умолчанию существующих миксинов. Например, переопределите вычисляемое свойство *breadcrumbs* для отключения хлебных крошек в представлении списка авторов.

```
import { ref } from 'vue';

spa.signals.once("allViews.created", ({ views }) => {
    const AuthorListView = views.get("/author/");
    AuthorListView.extendStore((store) => {
        return {
            ...store,
```

(continues on next page)

[419] https://v3.vuejs.org/guide/render-function.html

(продолжение с предыдущей страницы)

```
            breadcrumbs: ref([]),
        };
    });
});
```

Иногда вам может потребоваться скрыть страницу с деталями по какой-то причине, но при этом сохранить доступ ко всем действиям и подссылкам с страницы списка. Чтобы сделать это, также следует слушать сигнал *«allViews.created»* и изменить параметр *hidden* с значения *false* по умолчанию на *true*, например:

```
spa.signals.once('allViews.created', ({ views }) => {
    const authorView = views.get('/author/{id}/');
    authorView.hidden = true;
});
```

4.6 Изменение заголовка представления

Чтобы изменить заголовок и строку, отображаемую в хлебных крошках, измените свойство *title* представления или метод *getTitle* для более сложной логики.

```
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 Базовая конфигурация Webpack

Чтобы использовать webpack в вашем проекте, переименуйте *webpack.config.js.default* в *webpack.config.js*. Каждый проект, основанный на vst-utils, содержит *index.js* в каталоге */frontend_src/app/*. Этот файл предназначен для вашего кода. Запустите команду *yarn* для установки всех зависимостей. Затем выполните *yarn devBuild* из корневого каталога вашего проекта для сборки статических файлов. Последним шагом является добавление собранного файла в *SPA_STATIC* в *settings.py*.

```
SPA_STATIC += [
    {'priority': 101, 'type': 'js', 'name': '{AppName}/bundle/app.js', 'source':
↪'project_lib'},
]
```

Файл конфигурации Webpack позволяет добавлять дополнительные статические файлы. В *webpack.config.js* добавьте дополнительные записи

```
const config = {
  mode: setMode(),
  entry: {
    'app': entrypoints_dir + "/app/index.js" // default,
    'myapp': entrypoints_dir + "/app/myapp.js" // just added
  },
```

Выходные файлы будут собраны в каталоге *frontend_src/{AppName}/static/{AppName}/bundle*. Имя выходного файла соответствует имени записи в *config*. В приведенном выше примере выходные файлы будут называться *app.js* и *myapp.js*. Добавьте все эти файлы в *STATIC_SPA* в *settings.py*. Во время установки vstutils через *pip* фронтенд-код собирается автоматически, поэтому вам может потребоваться добавить каталог *bundle* в *gitignore*.

4.8 Хранилище страницы

У каждой страницы есть хранилище, которое можно получить глобально *app.store.page* или из компонента страницы с использованием *this.store*.

Метод представления *extendStore* можно использовать для добавления пользовательской логики в хранилище страницы.

```
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 Переопределение корневого компонента

Корневой компонент приложения можно переопределить с использованием сигнала *app.beforeInit*. Это может быть полезно, например, для изменения классов CSS макета, поведения кнопки назад или основных компонентов макета.

Пример настройки компонента боковой панели:

```
const CustomAppRoot = {
    components: { Sidebar: CustomSidebar },
    mixins: [spa.AppRoot],
};
spa.signals.once('app.beforeInit', ({ app }) => {
    app.appRootComponent = CustomAppRoot;
});
```

4.10 Перевод значений полей

Значения, отображаемые с использованием *FKField* или *ChoicesField*, могут быть переведены с использованием стандартных файлов перевода.

Ключ перевода должен быть определен как *:model:<ModelName>:<fieldName>:<value>*. Например:

```
TRANSLATION = {
    ':model:Category:name:Category 1': 'Категория 1',
}
```

Перевод значений может быть трудоемким, поскольку каждая модель на бэкенде обычно генерирует более одной модели на фронтенде. Для избежания этого добавьте атрибут *_translate_model = „Category“* к модели на бэкенде. Это сокращает

```
':model:Category:name:Category 1': 'Категория 1',
':model:OneCategory:name:Category 1': 'Категория 1',
':model:CategoryCreate:name:Category 1': 'Категория 1',
```

в

```
':model:Category:name:Category 1': 'Категория 1',
```

Для *FKField* используется имя связанной модели. И *fieldName* должно быть равно *viewField*.

4.11 Изменение действий или подссылок

Иногда использование только схемы для определения действий или подссылок недостаточно.

Например, у нас есть действие, которое делает пользователя суперпользователем (*/user/{id}/make_superuser/*), и мы хотим скрыть это действие, если пользователь уже является суперпользователем (*is_superuser* равно *true*). Сигнал *<${PATH}>filterActions* может быть использован для достижения такого результата.

```
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. *<${PATH}>filterActions* получает {actions, data}
2. *<${PATH}>filterSublinks* получает {sublinks, data}

Свойство *data* будет содержать данные экземпляра. Свойства *actions* и *sublinks* будут содержать массивы с элементами по умолчанию (не скрытыми действиями или подссылками), их можно изменить или полностью заменить.

4.12 Локальные настройки

Поля этой модели отображаются в левой боковой панели. Все данные из этой модели сохраняются в локальном хранилище браузера. Если вы хотите добавить другие варианты, вы можете сделать это, используя сигнал *beforeInit*, например:

```
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 Хранилище

Есть три способа сохранения данных:

- userSettingsStore - сохраняет данные на сервере. По умолчанию есть варианты изменения языка и кнопка включения/выключения темного режима. Данные для userSettingsStore поступают из схемы.
- localSettingsStore - сохраняет данные в локальном хранилище браузера. Здесь вы можете хранить свои собственные поля, как описано в *Локальные настройки*.
- store - хранит данные текущей страницы.

Чтобы использовать любое из этих хранилищ, вам нужно выполнить следующую команду: `app.[storeName]`, например: `app.userSettingsStore`.

Примечание: Если вы обращаетесь к userSettingsStore изнутри компонента, тогда вам нужно использовать `this.$app` вместо `app`.

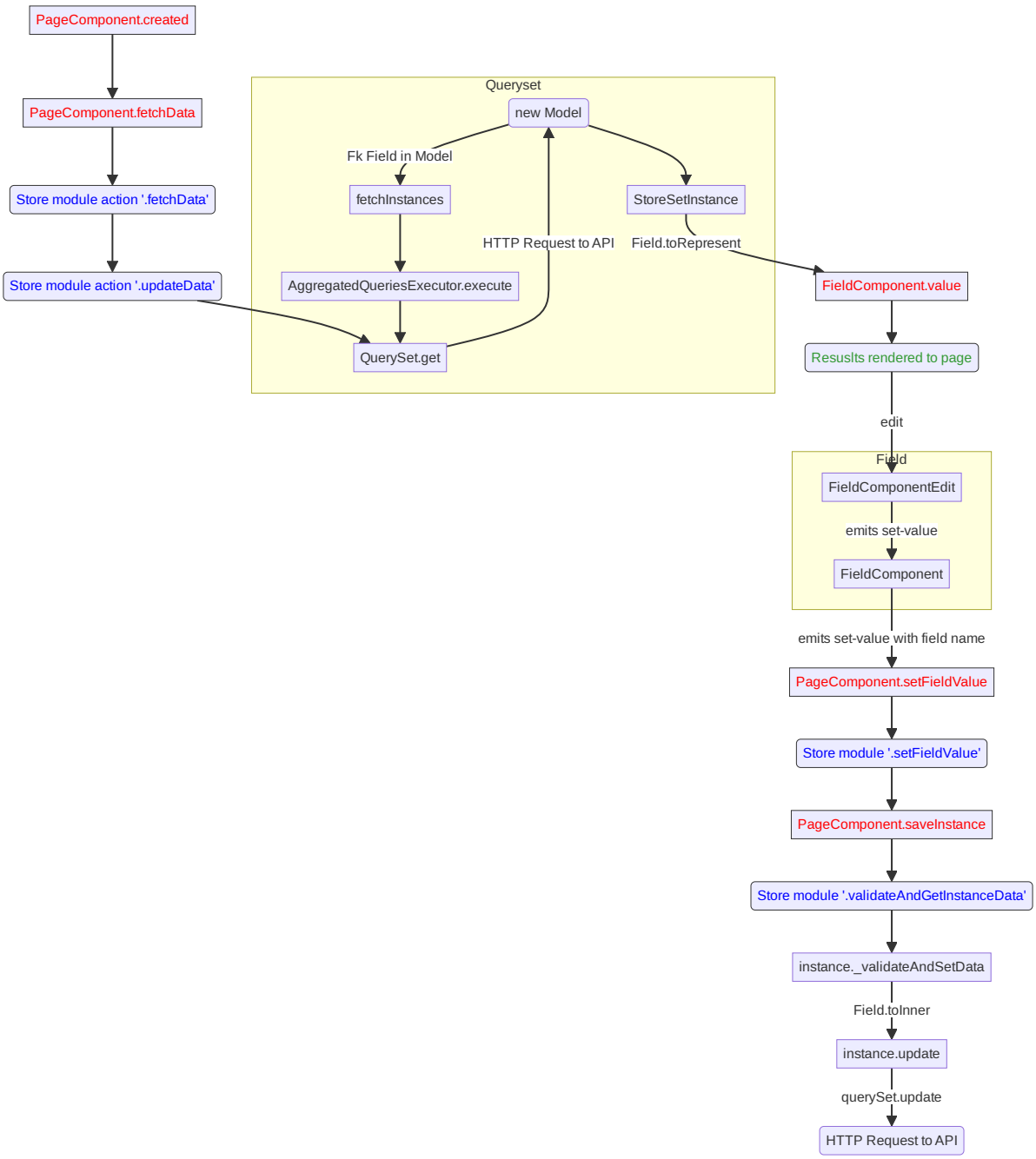
Из *app.store* вам может потребоваться:

- *vewsItems* и *viewItemsMap* - хранят информацию о родительских представлениях для этой страницы. Они используются, например, в хлебных крошках. Различие между ними заключается только в способе хранения информации: *viewItems* - это массив объектов, а *viewItemsMap* - это карта.
- *page* - сохраняет всю информацию о текущей странице.
- *title* - заголовок текущей страницы.

5 Документация по фронтенду

5.1 Схема API

Эта схема показывает, как данные проходят через приложение от и до API.



5.2 Сигналы

Система сигналов - это механизм, который VST Utils использует для настройки приложения.

Давайте посмотрим, как это работает.

Очень часто вам нужно что-то изменить после того, как произошло какое-то событие. Но как вы можете узнать о этом событии? А что, если вам нужно знать об этом событии в нескольких блоках кода?

Чтобы решить эту проблему, VST Utils использует систему сигналов, где:

- вы можете отправить какой-то сигнал, который сообщит всем подписчикам, что произошло какое-то событие, и передать некоторые данные/переменные из контекста, где произошло это событие;
- вы можете подписаться на какой-то сигнал, который уведомит вас о каком-то событии, и вы также можете передать некоторый обратный вызов (обработчик), который может делать что-то с данными/переменными, которые были переданы из контекста, где произошло событие.

5.2.1 Генерация сигнала

Чтобы сгенерировать сигнал, вам нужно написать следующее в своем коде:

```
tabSignal.emit(name_of_signal, context);
```

где:

- **name_of_signal** - строка, которая хранит имя сигнала (события);
- **context** - некоторая переменная любого типа, которая будет передана в обратный вызов (обработчик) во время подключения к этому сигналу.

Пример генерации сигнала:

```
let app = {
    name: 'example of app';
};

tabSignal.emit('app.created', app);
```

5.2.2 Подключение к сигналу

Чтобы подключиться к какому-то сигналу, вам нужно написать следующее в своем коде:

```
tabSignal.connect(name_of_signal, callback);
```

где:

- **name_of_signal** - строка, которая хранит имя сигнала (события);
- **callback** - функция, которая может выполнять действия с переменными, которые будут переданы из контекста события в этот обратный вызов в качестве аргументов.

Пример подключения к сигналу:

```
/* ... */
function callback(app) {
    app.title = 'example of app title';
}

tabSignal.connect('app.created', callback);
/* ... */
```

5.3 Список сигналов в VST Utils

VST Utils имеет несколько сигналов, которые генерируются во время работы приложения. Если вам нужно настроить что-то в вашем проекте, вы можете подписаться на эти сигналы и добавить функцию обратного вызова с желаемым поведением. Также вы можете генерировать свои сигналы в своем проекте.

5.3.1 openapi.loaded

Имя сигнала: «openapi.loaded».

Аргумент контекста: openapi - {object} - OpenAPI-схема, загруженная из API.

Описание: Этот сигнал генерируется после загрузки схемы OpenAPI. Вы можете использовать этот сигнал, если вам нужно что-то изменить в схеме OpenAPI, прежде чем она будет разобрана.

5.3.2 resource.loaded

Имя сигнала: «resource.loaded».

Аргумент контекста: Нет.

Описание: Этот сигнал генерируется после успешной загрузки всех статических файлов и их добавления на страницу.

5.3.3 app.version.updated

Имя сигнала: «app.version.updated».

Аргумент контекста: Нет.

Описание: Этот сигнал генерируется во время загрузки приложения, если VST Utils обнаруживает, что версия вашего проекта была обновлена.

5.3.4 app.beforeInitRouter

Имя сигнала: «app.beforeInitRouter».

Аргумент контекста: obj - {object} - Объект со следующей структурой: {routerConstructor: RouterConstructor}, где routerConstructor - это экземпляр RouterConstructor.

Описание: Этот сигнал генерируется после создания экземпляра RouterConstructor и перед созданием приложения.

5.3.5 app.beforeInit

Имя сигнала: «app.beforeInit».

Аргумент контекста: obj - {object} - Объект со следующей структурой: {app: app}, где app - это экземпляр класса App.

Описание: Этот сигнал генерируется после инициализации переменной приложения (разбор схемы OpenAPI, создание моделей и представлений), но перед тем, как приложение будет смонтировано на страницу.

5.3.6 app.afterInit

Имя сигнала: «app.afterInit».

Аргумент контекста: obj - {object} - Объект со следующей структурой: {app: app}, где app - это экземпляр класса App.

Описание: Этот сигнал генерируется после того, как приложение было смонтировано на страницу.

5.3.7 app.language.changed

Имя сигнала: «app.language.changed».

Аргумент контекста: obj - {object} - Объект со следующей структурой: {lang: lang}, где lang - это код примененного языка.

Описание: Этот сигнал генерируется после изменения языка интерфейса приложения.

5.3.8 models[model_name].fields.beforeInit

Имя сигнала: «models[» + model_name + «].fields.beforeInit». Например, для модели пользователя: «models[User].fields.beforeInit».

Контекстный аргумент: fields - {object} - Объект с парами ключ-значение, где ключ - имя поля, значение - объект с его опциями. На данный момент поле - просто объект с опциями, это не экземпляр guiFields.

Описание: Этот сигнал генерируется перед созданием экземпляров guiFields для полей модели.

5.3.9 models[model_name].fields.afterInit

Имя сигнала: «models[» + model_name + «].fields.afterInit». Например, для модели пользователя: «models[User].fields.afterInit».

Контекстный аргумент: fields - {object} - Объект с парами ключ-значение, где ключ - имя поля, значение - экземпляр guiFields.

Описание: Этот сигнал генерируется после создания экземпляров guiFields для полей модели.

5.3.10 models[model_name].created

Имя сигнала: «models[» + model_name + «].created». Например, для модели пользователя: «models[User].created».

Контекстный аргумент: obj - {object} - Объект со следующей структурой: {model: model}, где model - созданная модель.

Описание: Этот сигнал генерируется после создания объекта модели.

5.3.11 allModels.created

Имя сигнала: «allModels.created».

Контекстный аргумент: obj - {object} - Объект со следующей структурой: {models: models}, где models - объект, хранящий объекты моделей.

Описание: Этот сигнал генерируется после создания всех моделей.

5.3.12 allViews.created

Имя сигнала: «allViews.created».

Контекстный аргумент: obj - {object} - Объект со следующей структурой: {views: views}, где views - объект со всеми экземплярами представлений.

Описание: Этот сигнал генерируется после создания всех экземпляров представлений с установленными свойствами actions / child_links / multi_actions / operations / sublinks.

5.3.13 routes[name].created

Имя сигнала: «routes[» + name + «].created». Например, для представления `/user/`: «routes[/user/].created».

Контекстный аргумент: route - {object} - Объект со следующей структурой: {name: name, path: path, component: component}, где name - имя маршрута, path - шаблон пути маршрута, component - компонент, который будет отображаться для текущего маршрута.

Описание: Этот сигнал будет генерироваться после формирования маршрута и добавления его в список маршрутов.

5.3.14 allRoutes.created

Имя сигнала: «allRoutes.created».

Контекстный аргумент: routes - {array} - Массив объектов маршрутов со следующей структурой: {name: name, path: path, component: component}, где name - имя маршрута, path - шаблон пути маршрута, component - компонент, который будет отображаться для текущего маршрута.

Описание: Этот сигнал генерируется после формирования всех маршрутов и добавления их в список маршрутов.

5.3.15 <${PATH}>filterActions

Имя сигнала: «<${PATH}>filterActions».

Контекстный аргумент: obj - {actions: Object[], data} - Actions - массив объектов действий. Data представляет данные текущего экземпляра.

Описание: Этот сигнал будет выполнен для фильтрации действий.

5.3.16 <${PATH}>filterSublinks

Имя сигнала: «<${PATH}>filterSublinks».

Контекстный аргумент: obj - {sublinks: Object[], data} - Sublinks - массив объектов подссылок. Data представляет данные текущего экземпляра.

Описание: Этот сигнал будет выполнен для фильтрации подссылок.

5.4 Field Format

Файловые форматы

Часто при создании нового приложения разработчикам нужно создавать общие поля некоторых базовых типов и форматов (строка, булево, число и так далее). Создание каждый раз подобной функциональности довольно скучно и неэффективно, поэтому мы попытались решить эту проблему с помощью VST Utils.

VST Utils содержит набор встроенных полей наиболее распространенных типов и форматов, которые могут использоваться в различных случаях. Например, когда вам нужно добавить какое-то поле в веб-форму, которое должно скрывать значение вставленного значения, просто установите соответствующий формат поля на `password` вместо `string`, чтобы вместо фактических символов отображались звездочки.

Все доступные классы полей хранятся в переменной `guiFields`.На данный момент в VST Utils 44 формата полей:

- **base** - базовое поле, все остальные поля наследуются от этого поля;
- **string** - строковое поле для вставки и представления коротких значений „string“;
- **textarea** - строковое поле для вставки и представления длинных значений „string“;
- **number** - числовое поле для вставки и представления значений „number“;
- **integer** - числовое поле для вставки и представления значений формата „integer“;
- **int32** - числовое поле для вставки и представления значений формата „int32“;
- **int64** - числовое поле для вставки и представления значений формата „int64“;
- **double** - числовое поле для вставки и представления значений формата „double“;
- **float** - числовое поле для вставки и представления значений формата „float“;
- **boolean** - логическое поле для вставки и представления значений „boolean“;
- **choices** - строковое поле с жестким набором предопределенных значений, пользователь может выбрать только одно из доступных значений;
- **autocomplete** - строковое поле с набором предопределенных значений, пользователь может либо выбрать одно из доступных значений, либо ввести свое собственное значение;
- **password** - строковое поле, скрывающее вставленное значение знаками „*“
- **file** - строковое поле, способное считывать содержимое файла;
- **secretfile** - строковое поле, которое может считать содержимое файла, а затем скрыть его при отображении;
- **binfile** - строковое поле, которое может считать содержимое файла и преобразовывать его в формат „base64“;
- **namedbinfile** - поле в формате JSON, которое принимает и возвращает JSON с 2 свойствами: name (строка) - имя файла и content (строка base64) - содержание файла;

- **namedbinimage** - поле в формате JSON, которое принимает и возвращает JSON с 2 свойствами: name (строка) - имя изображения и content (строка base64) - содержание изображения;
- **multiplenamedbinimage** - поле в формате JSON, которое принимает и возвращает массив объектов, состоящих из 2 свойств: name (строка) - имя изображения и content (строка base64) - содержание изображения;
- **multiplenamedbinimage** - поле в формате JSON, которое принимает и возвращает массив объектов, состоящих из 2 свойств: name (строка) - имя изображения и content (строка base64) - содержание изображения;
- **text_paragraph** - строковое поле, представленное в виде текстового абзаца (без каких-либо вводов);
- **plain_text** - строковое поле, сохраняющее все непечатные символы во время представления;
- **html** - строковое поле, содержащее различные теги HTML и отображающее их во время представления;
- **date** - поле даты для вставки и представления значений „date“ в формате „YYYY-MM-DD“;
- **date_time** - поле даты для вставки и представления значений „date“ в формате „YYYY-MM-DD HH:mm“;
- **uptime** - строковое поле, которое преобразует продолжительность времени (количество секунд) в один из наиболее подходящих вариантов (23:59:59 / 01d 00:00:00 / 01m 30d 00:00:00 / 99y 11m 30d 22:23:24) в зависимости от его размера;
- **time_interval** - числовое поле, которое преобразует время из миллисекунд в секунды;
- **crontab** - строковое поле, которое имеет дополнительную форму для создания расписания в формате „crontab“;
- **json** - поле в формате JSON, во время представления использует другие guiFields для представления текущих свойств поля;
- **api_object** - поле, используемое для представления некоторого экземпляра модели из API (значение этого поля - все данные экземпляра модели). Это только для чтения поле;
- **fk** - поле, используемое для представления некоторого экземпляра модели из API (значение этого поля - первичный ключ экземпляра модели). Во время режима редактирования у этого поля есть строгий набор предопределенных значений для выбора;
- **fk_autocomplete** - поле, используемое для представления некоторого экземпляра модели из API (значение этого поля - первичный ключ экземпляра модели или некоторая строка). Во время режима редактирования пользователь может либо выбрать одно из предопределенных значений из списка автозаполнения, либо ввести свое собственное значение;
- **fk_multi_autocomplete** - поле, используемое для представления некоторого экземпляра модели из API (значение этого поля - первичный ключ экземпляра модели или некоторая строка). Во время режима редактирования пользователь может либо выбрать одно из предопределенных значений из модального окна, либо ввести свое собственное значение;
- **color** - строковое поле, которое сохраняет шестнадцатеричный код выбранного цвета;
- **inner_api_object** - поле, которое связано с полями другой модели;
- **api_data** - поле для представления некоторых данных из API;
- **dynamic** - поле, которое может изменять свой формат в зависимости от значений окружающих полей;
- **hidden** - поле, скрытое от представления;
- **form** - поле, объединяющее несколько других полей и сохраняющее их значения как один JSON, где key - имя поля формы, value - значение поля формы;
- **button** - специальное поле для формы, имитирующее кнопку в форме;
- **string_array** - поле, преобразующее массив строк в одну строку;

- **string_id** - строковое поле, предназначенное для использования в URL как ключ „id“. Он имеет дополнительную проверку, которая проверяет, что значение поля не равно некоторым другим ключам URL (new/ edit/ remove).

5.5 Настройка макета с использованием CSS

Если вам нужно настроить элементы с помощью CSS, у нас есть некоторая функциональность для этого. К корневым элементам `EntityView` (если он содержит *ModelField*), `ModelField`, `ListTableRow` и `MultiActions` применяются классы в зависимости от полей, которые они содержат. Классы формируются для полей типов «boolean» и «choices». Кроме того, классы применяются к кнопкам операций и ссылкам.

Правила генерации классов

- `EntityView, ModelField и ListTableRow` - *field-[field_name]-[field-value]*

  Пример:

  - *«field-active-true»* для модели, содержащей поле «boolean» с именем «active» и значением «true»
  - *«field-tariff_type-WAREHOUSE»* для модели, содержащей поле «choices» с именем «tariff_type» и значением «WAREHOUSE»

- **`MultiActions`** - ***selected__field-[field_name]-[field-value]***

  Пример:
  «selected__field-tariff_type-WAREHOUSE» и *«selected__field-tariff_type-SLIDE»*, если выбраны 2 строки `ListTableRow`, содержащие поле «choices» с именем «tariff_type» и значениями «WAREHOUSE» и «SLIDE» соответственно.

- **`Operation`** - ***operation__[operation_name]***

  Предупреждение
  Если вы скрываете операции с использованием классов CSS и, например, все действия были скрыты, то кнопка выпадающего списка действий все равно будет видна.

  Для более тщательного контроля над действиями и подссылками см. *Изменение действий или подссылок*

  Пример:
  operation__pickup_point, если кнопка операции или ссылка имеют имя *pickup_point*

На основе этих классов вы можете изменять стили различных элементов.

Несколько вариантов использования:

- Если вам нужно скрыть кнопку для действия «change_category» на странице подробной информации о продукте, когда продукт неактивен, вы можете сделать это, добавив селектор CSS:

```
.field-status-true .operation__change_category {
    display: none;
}
```

- Скрыть кнопку для действия «remove» в меню `MultiActions`, если выбран хотя бы один продукт со статусом «active»:

```
.selected__field-status-true .operation__remove {
    display: none;
}
```

- Если вам нужно изменить *цвет фона* на красный для заказа со статусом «CANCELLED» на компоненте `ListView`, сделайте следующее:

```
.item-row.field-status-CANCELLED {
    background-color: red;
}
```

  В этом случае вам нужно использовать дополнительный класс «item-row» (Используется для например, вы можете выбрать другой) для указания элемента, который будет выбран в селекторе, потому что класс «field-status-CANCELLED» добавляется в различные места на странице.

Содержание модулей Python

v

Алфавитный указатель

E

F

G

H

S

T

U

V

W