GeekBrains

Django REST Framework

Система версий API. Документация для API

На этом уроке

1. Узнаем, для чего используется система версий API.
2. Узнаем, какие возможности есть в DRF для контроля версий.
3. Научимся создавать и вести несколько версий API.
4. Узнаем, как создаётся документация к API.
5. Научимся использовать Swagger для создания документации.

Оглавление

Система версий в DRF

Введение

Очень часто, особенно в больших проектах, нам требуется поддерживать несколько версий API. Это позволяет сделать гибкую систему и не бояться, что при изменении какой-то части API клиенты, которые его используют, перестанут работать.

Описание работы системы версий можно найти в официальной документации DRF. На этом занятии мы разберём, как и когда использовать предоставленные нам средства.

В официальной документации говорится, что мы можем получить версию API, но как её использовать — это мы должны решить сами. Приводится следующий пример:

```
def get_serializer_class(self):
    if self.request.version == 'v1':
        return AccountSerializerVersion1
    return AccountSerializer
```

Нам предлагают получать номер версии из `request.version`, где `request` — это объект запроса в DRF. Далее мы можем использовать эту версию для различных `serializers` или каким-то другим образом.

Разберём этот важный момент подробнее:

1. DRF предоставляет механизмы указания разных версий. Что делать с номером версии, решаем мы сами.
2. Скорее всего, изменения версий будут касаться `serializers` и `views`. Разные `serializers` позволят нам иметь разное представление данных в зависимости от версии. А разные `views` позволяют по-разному обрабатывать различные виды запросов.

Далее рассмотрим на примере нашего демонстрационного проекта:

1. Какие средства у нас есть для указания разных версий API.
2. Какие плюсы и минусы у каждого из вариантов.
3. Как использовать эти варианты для нашего проекта.

Значение номера версии

Прежде чем мы перейдём к коду, рассмотрим, как нумеруются версии. Обычно версия программного обеспечения состоит из трёх цифр. Например, 3.1.2 — эти цифры разделены точками. Каждая часть номера имеет своё значение.

Последняя цифра в номере версии, например x.x.2, говорит, что вышло исправление некоторых ошибок. Сохраняется полная совместимость с предыдущей версией.

Вторая цифра в номере версии, например x.1.x, говорит, что в версии не было изменений, но был добавлен новый функционал. Сохраняется совместимость со старой версией. Клиенты, работающие с версией x.0.x, могут без проблем перейти на версию x.1.x, но обратный переход не всегда возможен.

Первая цифра в номере версии, например 3.x.x, говорит об изменениях в версии. Эта версия становится несовместима со старой.

Для REST API обычно бывает достаточно двух цифр, а иногда даже одной. При использовании двух цифр, например 3.0, мы можем учесть факт добавления нового функционала: например, в сериализаторе добавилось некоторое поле. При использовании одной цифры, например v2, мы создаём новую версию, только когда функционал изменился и больше не совместим со старой версией.

Указание версии в части URL-адреса

Основные варианты указания версии в DRF:

- в части URL-адреса;
- в параметре URL-адреса;
- в заголовках запроса.

Рассмотрим каждый из этих вариантов, начиная с первого.

В нашем проекте добавим API для вывода списка пользователей (модель User).

Для этого создадим новое приложение userapp:

```
python manage.py startapp userapp
```

terminal

В нём создадим файл serializers.py со следующим кодом:

```
from django.contrib.auth.models import User
from rest_framework import serializers


class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('username', 'email')
```

```
class UserSerializerWithFullName(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('username', 'email', 'first_name', 'last_name')
```

/userapp/serializers.py

Мы создали два сериализатора с разным списком полей. Будем их использовать для разных версий API.

Далее в файле views.py напишем следующий код:

```
from rest_framework import generics
from django.contrib.auth.models import User
from .serializers import UserSerializer, UserSerializerWithFullName


class UserListAPIView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer

    def get_serializer_class(self):
        if self.request.version == '0.2':
            return UserSerializerWithFullName
        return UserSerializer
```

/userapp/views.py

Если версия API 0.2, то мы будем использовать `UserSerializerWithFullName`, во всех остальных случаях — `UserSerializer`.

Теперь нам нужно выбрать, каким способом мы будем передавать версию.

UrlPathVersioning

Первый вариант — использовать класс UrlPathVersioning. Для этого в settings.py мы указываем следующую настройку:

```
REST_FRAMEWORK = {
    'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.URLPathVersioning',
...
}
```

/library/settings.py

Внимание! *По умолчанию система версий не включена, и `request.version` вернёт `None`.*

При использовании `UrlPathVersioning` мы можем передать номер версии в URL-адресе. В urls.py добавим следующий код:

```
urlpatterns = [
    ...
    re_path(r'^api/(?P<version>\d\.\d)/users/$', UserListAPIView.as_view()),
...
```

/library/urls.py

С помощью регулярного выражения мы указываем параметр `version`, который имеет вид x.x. Версия состоит из двух цифр с точкой между ними.

Теперь, чтобы работать с версией API 0.2, мы отправляем запрос на адрес:

```
http://127.0.0.1:8000/api/0.2/users/
```

NamespaceVersioning

NamespaceVersioning похож на предыдущий вариант, но версию мы указываем как `namespace`, связанный с группой адресов.

Для демонстрации в settings.py выберем `NamespaceVersioning`:

```
REST_FRAMEWORK = {
    #'DEFAULT_VERSIONING_CLASS':
'rest_framework.versioning.URLPathVersioning',
    'DEFAULT_VERSIONING_CLASS': 'rest_framework.versioning.NamespaceVersioning',
...
```

/library/settings.py

В приложении userapp локально создадим файл urls.py со следующим кодом:

```
from django.urls import path
from .views import UserListAPIView

app_name = 'userapp'
urlpatterns = [
    path('', UserListAPIView.as_view()),
]
```

/userapp/urls.py

Мы будем использовать тот же пример с `UserListAPIView`.

Далее в urls.py всего проекта добавим следующий код:

```
urlpatterns = [
    ...
    path('api/users/0.1', include('userapp.urls', namespace='0.1')),
    path('api/users/0.2', include('userapp.urls', namespace='0.2')),
```

/library/urls.py

В параметре `namespace` мы указываем версию API. Именно это значение мы будем получать в `request.version`. Функция `include` позволяет выбрать группу адресов и для них использовать разные версии.

Чтобы проверить работу нашего кода, перейдём на адрес:

```
http://127.0.0.1:8000/api/users/0.1
```

и

```
http://127.0.0.1:8000/api/users/0.2
```

Мы получим разные ответы от сервера.

HostNameVersioning

Класс HostNameVersioning позволяет указать версию в имени хоста, например http://v1.example.com/bookings/. Он используется довольно редко, и для его работы требуется настроенный «боевой» веб-сервер.

Указание версии в параметре URL-адреса

Альтернатива указанию версии в части адреса — указание версии в качестве его параметра.

Для этого используется класс QueryParameterVersioning. Для демонстрации его работы внесём изменения в наш проект.

Допустим, по какой-то причине мы решили, что не будем выводить год рождения автора в новой версии API. В файле serializers.py приложения mainapp добавим `Serializer` для этой цели. Код примет следующий вид:

```
from rest_framework import serializers
from .models import Author, Book

class AuthorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Author
        fields = '__all__'

class AuthorSerializerBase(serializers.ModelSerializer):
    class Meta:
        model = Author
        fields = ('name',)

class BookSerializerBase(serializers.ModelSerializer):
    # 10
    class Meta:
        model = Book
        fields = '__all__'

class BookSerializer(serializers.ModelSerializer):
    author = AuthorSerializer()
    class Meta:
        model = Book
        fields = '__all__'
```

/mainapp/serializers.py

Мы добавили `AuthorSerializerBase` с полем `name`.

Далее во views.py изменим код следующим образом:

```
from rest_framework import viewsets, permissions
from .models import Author, Book
from .serializers import AuthorSerializer, AuthorSerializerBase, BookSerializer,
BookSerializerBase


class AuthorViewSet(viewsets.ModelViewSet):
```

```
    serializer_class = AuthorSerializer
    queryset = Author.objects.all()

    def get_serializer_class(self):
        if self.request.version == '2.0':
            return AuthorSerializerBase
        return AuthorSerializer


class BookViewSet(viewsets.ModelViewSet):
    serializer_class = BookSerializer
    queryset = Book.objects.all()

    def get_serializer_class(self):
        if self.request.method in ['GET']:
            return BookSerializer
        return BookSerializerBase
```

/mainapp/views.py

В `AuthorViewSet` мы добавили проверку версии API и возврат разных сериализаторов.

Теперь в settings.py включим `QueryParameterVersioning`:

```
REST_FRAMEWORK = {
    #'DEFAULT_VERSIONING_CLASS':
'rest_framework.versioning.URLPathVersioning',
    #'DEFAULT_VERSIONING_CLASS':
'rest_framework.versioning.NamespaceVersioning',
    'DEFAULT_VERSIONING_CLASS':
'rest_framework.versioning.QueryParameterVersioning',
...
```

/library/settings.py

Чтобы проверить работоспособность нашего кода, перейдём по адресу:

```
http://127.0.0.1:8000/api/authors/?version=2.0
```

Указание версии в заголовках запроса

Этот способ считается оптимальным. Для его использования применяется класс AcceptHeaderVersioning.

Включим этот класс в settings.py:

```
REST_FRAMEWORK = {
    #'DEFAULT_VERSIONING_CLASS':
'rest_framework.versioning.URLPathVersioning',
    #'DEFAULT_VERSIONING_CLASS':
'rest_framework.versioning.NamespaceVersioning',
    # 'DEFAULT_VERSIONING_CLASS':
'rest_framework.versioning.QueryParameterVersioning',
    'DEFAULT_VERSIONING_CLASS':
'rest_framework.versioning.AcceptHeaderVersioning,
...
```

/library/settings.py

Для проверки работоспособности нам понадобится отправить запрос с заголовками. Это можно сделать из клиентской части приложения с помощью Axios или, например, с помощью библиотеки Requests:

```
import requests


response = requests.get('http://127.0.0.1:8000/api/authors/') # [{'id': 1,
'name': 'Грин', 'birthday_year': 1880}, {'id': 2, 'name': 'Пушкин',
'birthday_year': 1799}]
print(response.json())

response = requests.get('http://127.0.0.1:8000/api/authors/', headers={'Accept':
'application/json; version=2.0'}) # [{'name': 'Грин'}, {'name': 'Пушкин'}]
print(response.json())
```

В первом случае без указания версии в заголовках мы получаем один ответ:

```
headers={'Accept': 'application/json; version=2.0'}
```

А при указании версии ответ не содержит поля `birthday_year`.

Плюсы, минусы и варианты применения

Внутри URL-адреса

Позволяет чётко сформировать адрес, содержащий версию API. Это делает запросы более строгими. `UrlPathVersioning` подходит для небольших проектов, `NamespaceVersioning` — для больших.

Минусы этого способа: малая гибкость и большое дублирование кода.

В параметрах URL-адреса

Это удобный и гибкий способ, который используется во многих API. Он позволяет указать конкретную версию API, если она нам нужна, а без указания — придерживаться версии по умолчанию.

Минус — дополнительный параметр в адресе запроса.

В заголовках запроса

Такой же гибкий вариант, как и предыдущий, при этом мы не передаём в адрес дополнительные параметры.

Минус — трудность отправки заголовков вместе с запросом. Например, мы не можем протестировать такой API из браузера.

Документация для API

Создание документации — важная часть разработки любого программного обеспечения. Для систем с REST API документация особенно важна, так как с API обычно работают несколько клиентов, которые не имеют доступа к нашему коду. Чтобы узнать, какими возможностями обладает наш API, разработчики этих клиентов (например, frontend-разработчики) будут изучать документацию.

Создание документации с нуля — трудная задача. К счастью, есть несколько стандартных форматов для документации и сторонние библиотеки для интеграции этих форматов в наш DRF-проект.

Форматы представления документации

Спецификация OpenAPI

В разделе официального сайта DRF, посвящённом созданию документации, используется OpenAPI-схема, которая впоследствии взаимодействует со Swagger и ReDoc. Рассмотрим этот момент подробнее.

В самом DRF есть возможность создания OpenAPI-спецификации. Это описано в разделе Schemas официальной документации.

OpenAPI-спецификация представляет собой спецификацию для создания интерфейса взаимодействия между системами. OpenAPI рассматривается как универсальный интерфейс для пользователей (клиентов) по взаимодействию с сервисами (серверами).

Простыми словами, это способ описания нашего API в определённом формате, на основе которого можно создать документацию к нему, а иногда и создать код самого API.

Swagger и ReDoc

После создания схемы OpenAPI можно использовать её для генерации документации. Удобно создавать документацию с помощью Swagger и ReDoc.

Пример использования схем и Swagger.

Swagger и ReDoc — сервисы, которые по-разному используют созданную спецификацию и по-разному отображают документацию пользователя.

drf-yasg

В большинстве случаев нам не нужно отдельно создавать схему и шаблон для использования Swagger. Есть сторонние библиотеки, которые позволяют делать это быстро и удобно.

Наиболее популярна библиотека drf-yasg, её рекомендуют сами разработчики DRF. Воспользуемся её возможностями для динамического создания документации на основе нашего API.

Установим библиотеку:

```
pip install -U drf-yasg
```

Далее подключим приложение в `INSTALLED_APPS` проекта:

```
INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',  # required for serving swagger ui's css/js
```

```
files
   'drf_yasg',
   ...
]
```

/library/settings.py

После этого в файле urls.py проекта создадим представление для схемы OpenAPI:

```
from drf_yasg.views import get_schema_view
from drf_yasg import openapi
...

schema_view = get_schema_view(
   openapi.Info(
      title="Library",
      default_version='0.1',
      description="Documentation to out project",
      contact=openapi.Contact(email="admin@admin.local"),
      license=openapi.License(name="MIT License"),
   ),
   public=True,
   permission_classes=[permissions.AllowAny],
)
...
```

/library/urls.py

Значения переданных параметров интуитивно понятны. Отдельно выделим параметр `permission_classes`. Он позволяет задать права на документацию.

Другие настройки и возможности описаны в официальной документации drf-yasg.

После создания представления схемы мы можем сгенерировать для неё разные виды документации. Добавим в файл urls.py следующий код:

```
urlpatterns = [
    ...
    re_path(r'^swagger(?P<format>\.json|\.yaml)$',
schema_view.without_ui(cache_timeout=0), name='schema-json'),
    path('swagger/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
    path('redoc/', schema_view.with_ui('redoc', cache_timeout=0),
name='schema-redoc'),
]
```

/library/urls.py

Мы создали три адреса для отображения документации.

Первый адрес:

```
re_path(r'^swagger(?P<format>\.json|\.yaml)$',
schema_view.without_ui(cache_timeout=0), name='schema-json'),
```

Отображает документацию в формате JSON или YAML. На скриншоте представлен фрагмент ответа.


```
{"swagger": "2.0", "info": {"title": "Library", "description": "Documentation to out project", "contact": {"email
"/", "consumes": ["application/json"], "produces": ["application/json"], "securityDefinitions": {"Basic": {"type'
"parameters": [{"name": "data", "in": "body", "required": true, "schema": {"$ref": "#/definitions/AuthToken"}}],
"/api/authors/": {"get": {"operationId": "api_authors_list", "description": "", "parameters": [], "responses": {"
{"operationId": "api_authors_create", "description": "", "parameters": [{"name": "data", "in": "body", "required'
"tags": ["api"]}, "parameters": []}, "/api/authors/{id}/": {"get": {"operationId": "api_authors_read", "descripti
{"operationId": "api_authors_update", "description": "", "parameters": [{"name": "data", "in": "body", "required'
"tags": ["api"]}, "patch": {"operationId": "api_authors_partial_update", "description": "", "parameters": [{"name
{"$ref": "#/definitions/Author"}}}, "tags": ["api"]}, "delete": {"operationId": "api_authors_delete", "descriptio
"description": "A unique integer value identifying this author.", "required": true, "type": "integer"}]}, "/api/b
{"type": "array", "items": {"$ref": "#/definitions/Book"}}}}, "tags": ["api"]}, "post": {"operationId": "api_book
/BookSerializerBase"}}], "responses": {"201": {"description": "", "schema": {"$ref": "#/definitions/BookSerialize
"parameters": [], "responses": {"200": {"description": "", "schema": {"$ref": "#/definitions/Book"}}}, "tags": ["
"schema": {"$ref": "#/definitions/BookSerializerBase"}}], "responses": {"200": {"description": "", "schema": {"$r
"parameters": [{"name": "data", "in": "body", "required": true, "schema": {"$ref": "#/definitions/BookSerializerB
{"operationId": "api_books_delete", "description": "", "parameters": [], "responses": {"204": {"description": ""
true, "type": "integer"}]}, "/api/users/0.1": {"get": {"operationId": "api_users_0.1_list", "description": "", "p
["api"]}, "parameters": []}, "/api/users/0.2": {"get": {"operationId": "api_users_0.2_list", "description": "", "
["api"]}, "parameters": []}, "/api/{version}/users/": {"get": {"operationId": "api_users_list", "description": ""
["api"]}, "parameters": [{"name": "version", "in": "path", "required": true, "type": "string"}]}}, "definitions":
"minLength": 1}, "password": {"title": "Password", "type": "string", "minLength": 1}, "token": {"title": "Token",
{"id": {"title": "ID", "type": "integer", "readOnly": true}, "name": {"title": "Name", "type": "string", "maxLeng
"object", "properties": {"id": {"title": "ID", "type": "integer", "readOnly": true}, "author": {"$ref": "#/defini
"author"], "type": "object", "properties": {"id": {"title": "ID", "type": "integer", "readOnly": true}, "name": {
{"required": ["username"], "type": "object", "properties": {"username": {"title": "Username", "description": "Rec
"minLength": 1}, "email": {"title": "Email address", "type": "string", "format": "email", "maxLength": 254}}}}}
```


Эти данные служат для машинной обработки документации.

Второй адрес:

```
path('swagger/', schema_view.with_ui('swagger', cache_timeout=0),
name='schema-swagger-ui'),
```

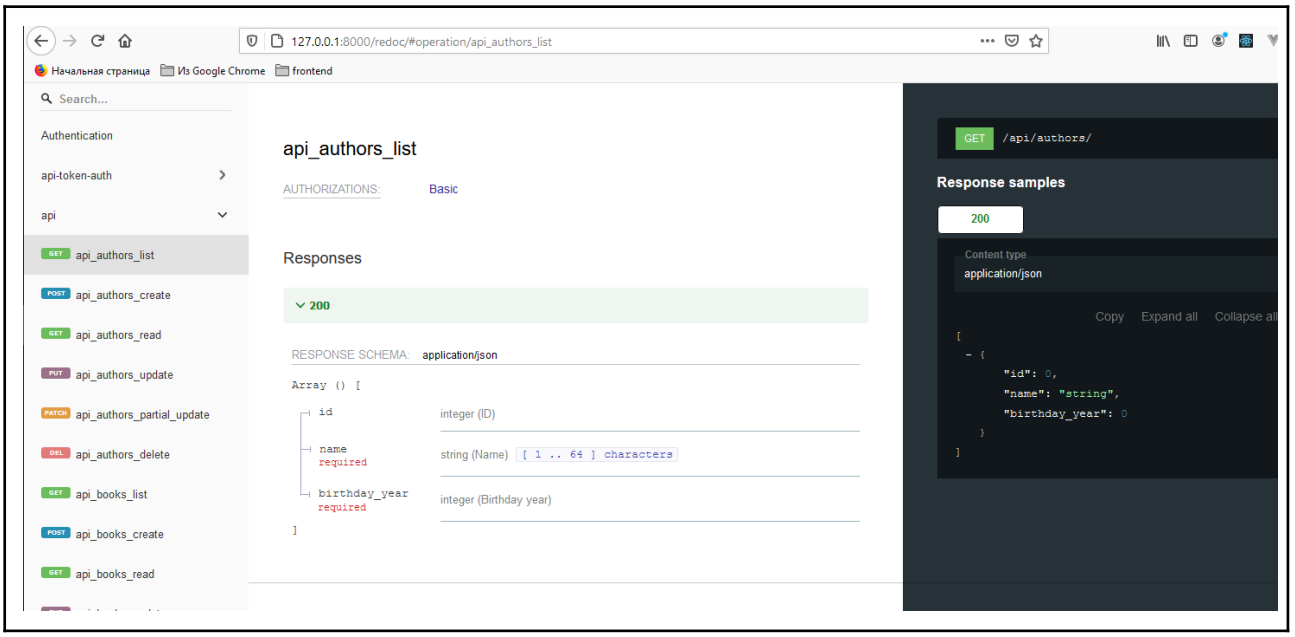
Формирует документацию с помощью Swagger. В браузере мы увидим удобную документацию по всему нашему API. Фрагмент представлен на скриншоте:

Эту документацию удобно использовать для создания клиента для нашего API.

Третий адрес:

```
 path('redoc/', schema_view.with_ui('redoc', cache_timeout=0),
name='schema-redoc'),
```

Тоже предоставляет удобную документацию, но уже в формате ReDoc:



Обычно достаточно машинного представления и какого-то одного варианта — Swagger или Redoc — для человеческого представления документации.

Резюме

Для создания документации необходимо создать спецификацию OpenAPI и после этого использовать Swagger или ReDoc. Это можно сделать как отдельно, так и при помощи сторонних библиотек.

На этом занятии мы рассмотрели систему версий в DRF и способы создания документации для REST API. Это позволит нам создавать гибкие API с удобной документацией по их использованию.

Глоссарий

OpenAPI (с англ. — «спецификация OpenAPI», изначально известная как Swagger Specification) — формализованная спецификация и экосистема множества инструментов, предоставляющая интерфейс между frontend-системами, кодом библиотек низкого уровня и коммерческими решениями в виде API. Построена таким образом, что не зависит от языков программирования и удобна в использовании как человеком, так и машиной.

OpenAPI рассматривается как универсальный интерфейс для пользователей (клиентов) по взаимодействию с сервисами (серверами). Если спроектирована спецификация для некоторого сервиса, то на её основании можно генерировать исходный код для библиотек клиентских приложений, текстовую документацию для пользователей, варианты тестирования и др. Для этих действий есть большой набор инструментов для различных языков программирования и платформ.

ReDoc — генератор документации на основе OpenAPI-спецификации.

Дополнительные материалы

1. Официальный сайт Swagger.
2. Документация drf-yasg.
3. ReDoc — GitHub.
4. Документация DRF.
5. Schemas DRF.
6. Статья по документированию микросервисов.
7. Статья про Swagger.
8. Дополнительные параметры drf-yasg.

Практическое задание

Использование системы версий. Создание документации.

В этой самостоятельной работе мы тренируем умения:

- тестировать REST API;
- создавать тестовые данные.

Смысл: использовать системы версий в проектах с DRF. Написать документацию для API.

Последовательность действий

1) Создать новую версию API в проекте, в которой у модели пользователя будут доступны поля `is_superuser`, `is_staff`. Таким образом, проект будет поддерживать две версии API.
2) Создать документацию для API, используя drf-yasg.
3) * Создать часть документации Swagger и/или ReDoc без использования сторонних библиотек. Можно использовать минимальные примеры из стандартной документации Swagger и ReDoc.