DRF: сериализация, ViewSet, роутеры | Я.Шпора

Django REST framework (DRF) — это набор инструментов для создания веб-сервисов и API на основе фреймворка Django.

Сериализация и десериализация данных

Сериализация — преобразование python-объекта в общепринятый формат данных для обмена информации через API, например, в JSON.

Например, такой объект…

```
post = Post(
    id=87,
    author='Робинзон Крузо',
    text='23 ноября. Закончил работу над лопатой и корытом.',
    pub_date='1659-11-23T18:02:33.123543Z'
)
```

…можно сериализовать в такой JSON:

```
{
    "id": 87,
    "author": "Робинзон Крузо",
    "text": "23 ноября. Закончил работу над лопатой и корытом.",
    "pub_date": "1659-11-23T18:02:33.123543Z"
}
```

Десериализация — процесс преобразования JSON данных в python-объект. При десериализации данные могут проверяться на корректность — **валидироваться**.

Из такого JSON…

```
{
    "author": "Робинзон Крузо",
    "text": "24 декабря. Всю ночь и весь день шёл проливной дождь
```

```
    "pub_date": "1659-12-24T21:14:56.123543Z"
}
```

…можно создать такой python-объект:

```
post = Post(
    author='Робинзон Крузо',
    text='24 декабря. Всю ночь и весь день шёл проливной дождь.',
    pub_date='1659-12-24T21:14:56.123543Z'
)
```

В Django REST Framework есть классы, выполняющие все три операции: сериализацию, валидацию и десериализацию. Эти классы называются **сериализаторы (serializers)**.

Сериализаторы могут работать с моделями Django и с обычными Python-классами.

- Для работы с обычными Python-классами сериализаторы наследуют от класса **Serializer**.
- Сериализаторы, работающие с моделями, наследуют от **ModelSerializer**.

Сериализатор для модели: ModelSerializer

Для работы с моделями сериализатор наследуется от класса ModelSerializer.

На примере модели `Comment`:

```
...

class Comment(models.Model):
    post = models.ForeignKey(Post, on_delete=models.CASCADE)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    text = models.TextField()
    created = models.DateTimeField('created', auto_now_add=True)
```

Сериализатор для модели `Comment`:

```
...
from rest_framework import serializers


class CommentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comment
        # Указываем поля модели, с которыми будет работать сериали
        # поля модели, не указанные в перечне, сериализатор будет
        # Для перечисления полей можно использовать список или кор
        fields = ('id', 'post', 'author', 'text', 'created')
```

Сериализатор для класса: Serializer

Сериализатор для класса работает аналогично сериализатору для модели. Разница в том, что **ModelSerializer** берёт описание полей из модели, а в классе **Serializer** нужно явным образом описать поля, их типы и параметры.

На примере класса `Book`:

```
class Book():
    def __init__(self, author, title, pub_year, genre):
        self.author = author
        self.title = title
        self.pub_year = pub_year
        self.genre = genre
```

Сериализатор для класса `Book`:

```
class BookSerializer(serializers.Serializer):
    # Описываем поля и их типы
    author = serializers.CharField(max_length=256)
    title = serializers.CharField(max_length=512)
    pub_year = serializers.IntegerField()
    genre = serializers.CharField(64)
```

Один и тот же класс-сериализатор в DRF можно применять как для сериализации, так и для десериализации данных. Логика его работы

выбирается автоматически, в зависимости от того, какой объект передан в сериализатор.

View-функции API

В Django REST Framework запрос к API передаётся нужному представлению в соответствии с адресами, перечисленными в файле *urls.py*.

Для настройки view-функции на работу с API в Django REST framework есть декоратор `@api_view`.

```
from rest_framework.decorators import api_view  # Импортировали де
from rest_framework.response import Response  # Импортировали клас


@api_view(['GET', 'POST'])  # Применили декоратор и указали разреш
def hello(request):
    # В ответ на POST-запрос вернём JSON с теми же данными,
    # которые получены в запросе.
    # Для этого в объект Response() передаём словарь request.data
    if request.method == 'POST':
        return Response({'message': 'Получены данные', 'data': req

    # В ответ на GET-запрос вернём JSON с текстовым сообщением.
    # Этот JSON будет создан из словаря, переданного в объект Resp
    return Response({'message': 'Это был GET-запрос!'})
```

Для обработки POST-запросов с помощью сериализаторов необходимо вызвать метод `.is_valid()`. Если валидация прошла успешно, то данные из запроса будут переданы в дальнейшую работу (в примере — сохранены в БД):

```
@api_view(['GET', 'POST'])
def cat_list(request):
    if request.method == 'POST':
        # Создаём объект сериализатора
        # и передаём в него данные из POST-запроса.
        serializer = CatSerializer(data=request.data)
        if serializer.is_valid():
            # Если полученные данные валидны —
```

```
            # сохраняем данные в базу через save().
            serializer.save()
            # Возвращаем JSON со всеми данными нового объекта
            # и статус-код 201.
            return Response(serializer.data, status=status.HTTP_20
        # Если данные не прошли валидацию —
        # возвращаем информацию об ошибках и соответствующий стату
        return Response(serializer.errors, status=status.HTTP_400_
    cats = Cat.objects.all()
    serializer = CatSerializer(cats, many=True)
    # При GET-запросе вернутся данные из запроса, сериализованные
    return Response(serializer.data)
```

Строка `Response(serializer.data)` сериализует данные из запроса в JSON и отправляет этот JSON ответом на запрос.

Чтобы сериализатор был готов принять список объектов, в конструктор сериализатора нужно передать именованный параметр `many=True`.

```
serializer = CatSerializer(data=request.data, many=True)
```

Чтобы сериализатор был готов вернуть список объектов, ему нужно передать этот список и указать именованный параметр `many=True`.

```
# Получаем все объекты модели.
cats = Cat.objects.all()
# Передаём queryset в конструктор сериализатора.
serializer = CatSerializer(cats, many=True)
```

Запись в БД с помощью сериализаторов

В результате вызова метода `serializer.save()` можно создать новую запись в БД или обновить существующую запись.

Сохранение записи может выглядеть так:

```
...
serializer = CatSerializer(data=request.data)
# Если вызвать serializer.save(), будет создана новая запись в БД
```

Для обновления записи в БД необходимо получить обновляемый объект из БД и передать его в сериализатор вместе с данными для обновления.

```
...
cat = Cat.objects.get(id=id)
serializer = CatSerializer(cat, data=request.data)
# Если вызвать serializer.save(), будет обновлён существующий экзе
```

При PATCH-запросах в сериализаторе необходимо указать именованный параметр `partial=True`.

```
serializer = CatSerializer(cat, data=request.data, partial=True)
```

View-классы API

View-классы обладают рядом преимуществ перед view-функциями:

- возможность применять готовый код для решения стандартных задач;
- наследование позволяет повторно использовать уже написанный код.

Низкоуровневые view-классы в DRF

Во view-классе, унаследованном от класса `APIView`, при получении GET-запроса будет автоматически вызван метод `get()`, а при получении POST-запроса — метод `post()`. Специальные методы есть и для всех остальных типов запросов.

По умолчанию эти методы не выполняют никаких действий, их нужно описывать самостоятельно.

```
from rest_framework.views import APIView


# Надо самостоятельно описать необходимые методы.
class MyAPIView(APIView):
    def get(self, request):
        ...

    def post(self, request):
```

```
        ...

    def put(self, request):
        ...

    def patch(self, request):
        ...

    def delete(self, request):
        ...
```

Generic Views: высокоуровневые view-классы

Для типовых действий, например, для вывода списка объектов или для получения объекта по id, удобнее использовать **высокоуровневые** view-классы, «дженерики» (англ. *Generic Views*): в них уже реализованы все механизмы, необходимые для решения стандартных задач.

На примере модели котика:

```
class Cat(models.Model):
    name = models.CharField(max_length=16)
    color = models.CharField(max_length=16)
    birth_year = models.IntegerField(blank=True, null=True)

    def __str__(self):
        return self.name
```

Универсальные дженерики

- Дженерик `ListCreateAPIView` реализует чтение коллекции объектов и создание одного объекта.
- Дженерик `RetrieveUpdateDestroyAPIView` реализует чтение, запись и удаление одного объекта модели.

Применение: классы `CatList` и `CatDetail` реализуют весь API CRUD для модели `Cat`.

```python
from rest_framework import generics

from .models import Cat
from .serializers import CatSerializer


class CatList(generics.ListCreateAPIView):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer


class CatDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
```

Специализированные Generic Views

Специализированные Generic Views:

- `ListAPIView` — выводит список объектов в ответ на GET-запрос (обрабатывает только GET-запросы);
- `RetrieveAPIView` — возвращает один объект (обрабатывает только GET-запросы);
- `CreateAPIView` — создаёт новый объект (обрабатывает только POST-запросы);
- `UpdateAPIView` — изменяет объект (обрабатывает только PUT- и PATCH-запросы);
- `DestroyAPIView` — удаляет объект (обрабатывает только DELETE-запросы).

ViewSet

Вьюсет (ViewSet)— это высокоуровневый view-класс, реализующий все операции CRUD; он может вернуть объект или список объектов, создать, изменить или удалить объекты.

Универсальный класс ModelViewSet

Класс `ModelViewSet` может выполнять любые операции CRUD с экземплярами модели. Для его работы необходимо указать queryset нужной модели и сериализатор:

```
from rest_framework import viewsets

from .models import Cat
from .serializers import CatSerializer


class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer
```

Несколько сериализаторов для одного вьюсета

В стандартном методе вьюсета `get_serializer_class()` можно определить, какой из доступных сериализаторов должен обрабатывать данные в зависимости от действия:

```
class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()

    ...

    def get_serializer_class(self):
        # Если запрошенное действие (action) — получение списка об
        if self.action == 'list':
            # ...то применяем CatListSerializer
            return CatListSerializer
        # А если запрошенное действие — не 'list', применяем CatSe
        return CatSerializer
```

Класс ReadOnlyModelViewSet: только чтение

Этот класс подобен классу `ModelViewSet`, но может только получать данные модели, а записывать и изменять — не может.

```
from rest_framework import viewsets

...

class CatViewSet(viewsets.ReadOnlyModelViewSet):
    ...
```

Расширение возможностей ViewSet

Список стандартных действий (англ. *actions*) во вьюсетах:

- *create*: создание экземпляра;
- *retrieve*: получение экземпляра;
- *list*: получение списка экземпляров;
- *update*: обновление экземпляра (все поля);
- *partial_update*: обновление экземпляра (только выбранные поля);
- *destroy*: удаление экземпляра.

Чтобы описать нестандартное действие, во вьюсет добавляют методы, которые оборачивают в декоратор `@action` («действие»). Этот декоратор настраивает метод и создаёт эндпоинты для этих действий.

Получим из базы пять белых котиков с наибольшими id — тех, что появились в базе последними. Для этого опишем метод `recent_white_cats` для вьюсета `CatViewSet`.

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

...

class CatViewSet(viewsets.ModelViewSet):
    queryset = Cat.objects.all()
    serializer_class = CatSerializer

    # Пишем метод, а в декораторе разрешим работу со списком объек
    # и переопределим URL на более презентабельный
```

```
    @action(detail=False, url_path='recent-white-cats')
    def recent_white_cats(self, request):
        # Из выборки с котиками белого цвета берём пять последних
        cats = Cat.objects.filter(color='White').order_by('id')[:5
        # Передадим queryset cats сериализатору
        # и разрешим работу со списком объектов
        serializer = self.get_serializer(cats, many=True)
        return Response(serializer.data)
```

Роутеры

При работе с view-классами и дженериками каждый эндпоинт отдельно описывается в *urls.py*. Для вьюсетов есть более удобный и экономичный инструмент — **роутеры.**

Роутер — это класс, который автоматически генерирует наборы путей для заданного вьюсета.

В DRF есть два стандартных роутера: `SimpleRouter` и `DefaultRouter`.

Для использования роутера `SimpleRouter` нужно:

- импортировать в файл *urls.py* класс `SimpleRouter`;
- создать экземпляр этого класса;
- «зарегистрировать» роутер — вызвать его метод `register()`;
- включить сгенерированные роутером пути в список `urlpatterns`:

```
# Файл urls.py
from django.urls import include, path
from rest_framework.routers import SimpleRouter  # Импортируем кла

router = SimpleRouter()  # Создаём объект роутера.
router.register('cats', CatViewSet)  # Регистрируем роутер.

urlpatterns = [
    ...
    # Все зарегистрированные в router пути доступны в router.urls
    path('', include(router.urls)),  # Подключаем пути роутера к u
```

```
    ...
]
```

В приведённом примере роутер `SimpleRouter` сгенерирует такой набор путей:

```
...
    path('cat/', ..., name='list'),
    path('cat/<int:pk>/', ..., name='detail'),
...
```

При работе с вьюсетами и роутерами может возникнуть ошибка *«не определён аргумент basename»*:

```
'basename' argument not specified, and could not automatically det
```

Параметр `basename` обязателен в тех случаях, когда queryset не указан во вьюсете явным образом, а определён через метод `get_queryset()`.

Речь идёт о необязательном аргументе роутера `basename`: в нём можно вручную указать префикс для параметра `name` в эндпоинтах, созданных роутером.

```
router.register('cats', CatViewSet, basename='tiger')
```

Такой роутер сгенерирует набор путей, у каждого из которых `name` будет начинаться с `tiger`:

```
urlpatterns = [
    path('cat/', ..., name='tiger-list'),
    path('cat/<int:pk>/', ..., name='tiger-detail'),
]
```

DefaultRouter — это расширенная версия `SimpleRouter`: он умеет всё то же, что и SimpleRouter, а в дополнение ко всему генерирует корневой эндпоинт `/`, GET-запрос к которому вернёт список ссылок на все ресурсы, доступные в API.

```
from rest_framework.routers import DefaultRouter

router = DefaultRouter()
```

Работа с `DefaultRouter` не отличается от `SimpleRouter`.