

Django REST framework: фильтрация, валидация и аутентификация



Адилет
Асанкожоев



Адилет Асанкожоев

Python-разработчик в Makers.kg





План занятия

1. [Фильтрация данных](#)
2. [Зачем нужна валидация данных](#)
3. [Валидация и сериализация](#)
4. [Сохранение данных](#)
5. [Аутентификация](#)



Фильтрация данных

Фильтрация данных

В базе данных могут лежат тысячи записей и миллионы записей, нам нужно показывать только часть из них.

Фильтры в DRF принято передать через GET-параметры запроса.

Установка django-filter:

```
$ pip install django-filter
```

Фильтрация данных

Затем чтобы включить фильтрацию, необходимо либо задать `DEFAULT_FILTER_BACKENDS` в настройках, либо задавать `filter_backends` для каждого `ViewSet`

```
class OrderViewSet(viewsets.ModelViewSet):  
    ...  
    filter_backends = [DjangoFilterBackend]  
    filterset_class = OrderFilter
```

Разберем подробнее, что такое `filterset_class`

Фильтрация данных

DRF предоставляет несколько вариантов для быстрой генерации фильтров, но лучше всего объявлять фильтры явно в виде классов:

```
from django_filters import rest_framework as filters
from orders.models import Order

class OrderFilter(filters.FilterSet):
    """Класс для определения фильтров."""

    id = filters.ModelMultipleChoiceFilter(to_field_name="id",
                                           queryset=Order.objects.all())
    amount_from = filters.NumberFilter(field_name="amount", lookup_expr="gte")
    amount_to = filters.NumberFilter(field_name="amount", lookup_expr="lte")

    class Meta:
        model = Order
        fields = ("id", "amount_from", "amount_to",)
```

Фильтрация данных

Благодаря этому фильтрация не перемешивается с бизнес-логикой и можно заводить сложные фильтры, например, используя методы:

<https://django-filter.readthedocs.io/en/stable/ref/filters.html?highlight=MultipleChoiceFilter#method>

Документация интеграции django-filter и DRF:

- <https://www.django-rest-framework.org/api-guide/filtering/>
- https://django-filter.readthedocs.io/en/stable/guide/rest_framework.html



Зачем нужна валидация данных

Валидация данных

Валидация данных – это проверка данных на backend'е. Нужно валидировать:

- **соответствие типов** – нельзя отправить строку вместо числа,
- **специфичные ограничения** – создать товар или заказ с отрицательной стоимостью,
- **бизнес-логику** – например, нельзя создать заказ без товаров.

Никакие данные не должны записываться в базу данных без валидации.



Валидация и сериализация

Для валидации данных в DRF принято использовать сериализаторы.
Модель:

```
class Product(models.Model):
    """Продукт в ассортименте."""
    ...

class ProductOrderPosition(models.Model):
    """Позиция в заказе."""
    product = models.ForeignKey(...)
    order = models.ForeignKey(
        "Order",
        related_name='positions',
        on_delete=models.CASCADE,
    )

    quantity = models.PositiveIntegerField(...)

class Order(models.Model):
    amount = models.DecimalField(...)
    notes = models.TextField(...)
    products = models.ManyToManyField(
        Product,
        through=ProductOrderPosition,
    )
```

Валидация в сериализаторе:

```
class ProductOrderPositionSerializer(serializers.Serializer):
    """Сериализатор для Product."""
    product = serializers.PrimaryKeyRelatedField(
        queryset=Product.objects.all(),
        required=True,
    )
    quantity = serializers.IntegerField(min_value=1, default=1)

class OrderSerializer(serializers.ModelSerializer):
    """Сериализатор для Order."""
    positions = ProductOrderPositionSerializer(many=True)

    class Meta:
        model = Order
        fields = ("id", "amount", "notes", "positions",)

    def validate_positions(self, value):
        if not value:
            raise serializers.ValidationError("Не указаны позиции заказа")
        product_ids = [item["product"].id for item in value]
        if len(product_ids) != len(set(product_ids)):
            raise serializers.ValidationError("Дублируются позиции в заказе")
```

Валидация полей

Валидацию для одного поля можно описать в методе `validate__<field_name>`.

Если нужна валидация для нескольких полей одновременно, используйте метод `validate`.

Пример ошибки:

```
HTTP 400 Bad Request
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
```

```
{
  "positions": [
    "Дублируются позиции в заказе"
  ]
}
```



Сохранение данных

Сохранение данных

В DRF принято описывать сохранение данных в сериализаторе.

Это похоже на прием, который используется в стандартных Джанговских формах: <https://docs.djangoproject.com/en/3.1/topics/forms/modelforms/>

В целом, DRF старается придерживаться паттернов и подходов, принятых в самом Django.

Валидация данных и сохранение будет выполняться автоматически, если вы используете `ModelViewSet`.

В противном случае, вы можете сохранить модель явно:

```
def create(self, request, *args, **kwargs):
    # слегка адаптированный код из исходников DRF
    serializer = self.get_serializer(data=request.data)
    serializer.is_valid(raise_exception=True)
    serializer.save()
    ...
    return Response(serializer.data, status=status.HTTP_201_CREATED)
```

Обратите внимание, что `create` — это метод `ViewSet`, который вызывается при POST-запросе.

При `PATCH` / `PUT` будет вызываться метод `update`. Подробнее в документации про `ModelViewSet`:

<https://www.django-rest-framework.org/api-guide/viewsets/#modelviewset>

Пример успешного ответа:

```
{
  "id": 5,
  "amount": "4000.00",
  "notes": "Не звонить утром",
  "positions": [
    {
      "product": 1,
      "quantity": 1
    },
    {
      "product": 2,
      "quantity": 2
    }
  ]
}
```


Область ответственности сериализатора:

- валидация данных
- представление данных
- создание объектов

Критически важно придерживаться соглашений и описывать логику там, где её принято описывать. Это позволит вам писать хорошо структурированный код, и с ним будет гораздо проще работать как вам, так и другим разработчикам.



Аутентификация



Аутентификация – проверка того, что пользователь является тем, за кого он себя выдает.

Авторизация – проверка прав пользователя.

Виды аутентификации

- **BasicAuthentication** – подходит только для тестирования
- **SessionAuthentication** – использует стандартный механизм аутентификации через использование сессии в Django, не рекомендуется для использования в микросервисной архитектуре
- **TokenAuthentication** – аутентификация через специальный токен в заголовках. Универсальный и хороший метод. Рассмотрим его подробнее.

TokenAuthentication

Чтобы использовать, необходимо подключить приложение:

```
INSTALLED_APPS = [  
    ...  
    'rest_framework.authtoken'  
]
```

И запустить миграцию: `manage.py migrate`

Токены можно создавать через админку Django. По-умолчанию, на каждого юзера создается только один токен (связь one-to-one). Если вам это не подходит, то вы можете переопределить у себя в коде модель Token и использовать ее.

Токены передаются через заголовок


Authorization: Token user_token_example

TokenAuthentication

Чтобы ваши ViewSet'ы стали использовать указанный метод аутентификации, пропишите `DEFAULT_AUTHENTICATION_CLASSES` в конфиге `REST_FRAMEWORK` или задайте `authentication_classes` на уровне ViewSet'a

<https://www.django-rest-framework.org/api-guide/authentication/#setting-the-authentication-scheme>

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
}
```

После прохождения аутентификации, ответственность клиента – сохранить токен и использовать его для последующих запросов.

JavaScript-приложение может сохранить токен в localStorage.

Поэтому ответственность сервера – предоставить адрес, по которому клиент может получить себе токен. DRF предоставляет встроенную view-функцию для этого:

```
from rest_framework.authtoken import views

urlpatterns += [
    url(r'^api-token-auth/', views.obtain_auth_token)
]
```

В зависимости от ваших потребностей, вы ее можете переопределить. Например, чтобы поддерживать несколько токенов для одного пользователя или срок жизни токена.

Проверка прав

Решается с помощью атрибута `permission_classes` во `ViewSet` или с помощью декоратора `permission_classes`:

```
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated

@api_view(['GET'])
@permission_classes([IsAuthenticated])
def example_view(request, format=None):
    """Ваша view-функция, доступная только залогинившимся
    пользователям."""
```

Итоги

Сегодня на занятии мы разобрали:

- Как фильтровать данные
- Как валидировать и сохранять данные
- Как проверять доступ и проводить аутентификацию пользователей



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

 [Адилет Асанкожоев](#)

Адилет Асанкожоев