CRUD в DRF

План занятия

1. CRUD
2. ViewSet и роутеры
3. Сериализаторы, валидация и обновление данных
4. Фильтрация данных
5. Пагинация данных

CRUD

CRUD

CRUD — это аббревиатура для Create-Read-Update-Delete.

Ей обозначают **логику** для операций создания, чтения, обновления и удаления сущностей

CRUD

CRUD в REST API реализуется через 6 запросов:

Примеры маршрутов		
GET	/api/orders/	получить все заказы
POST	/api/orders/	создать новый заказ
GET	/api/orders/<id>/	получить конкретный заказ по id
PATCH	/api/orders/<id>/	изменить конкретный заказ по id
PUT	/api/orders/<id>/	заменить конкретный заказ по id
DELETE	/api/orders/<id>/	удалить конкретный заказ по id

ViewSet и роутеры

View

Class-based view

Методы класса описывают обработчиков для HTTP-методов

Viewset

Каждый метод класса описывает обработку соответствующего запроса на ресурс:

- Спрос на специалистов на рынке труда
- создание
- получение
- список
- обновление
- удаление

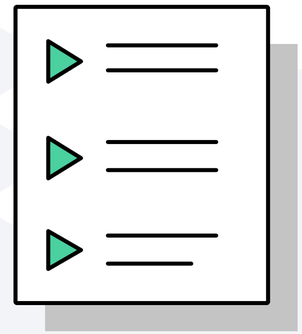
Если какой-то метод не реализован, Django будет возвращать **ошибку 405**

ViewSet

ViewSet — встроенная в **DRF** батарейка для стандартной CRUD (create-read-update-delete)-логики на моделях.

Конфигурация ViewSet описывается с помощью **атрибутов и методов класса**.

Мы описываем, что достаем и в каком формате отдаём, с помощью атрибутов queryset и serializer



ViewSet

```
class SomeViewSet(viewsets.ViewSet):
  def list(self, request):
    pass  # R - прочитать все
  def create(self, request):
    pass  # C - создать
  def retrieve(self, request, pk=None):
    pass  # R - прочитать по pk
  def update(self, request, pk=None):
    pass  # U - обновить (полностью) по pk
  def partial_update(self, request, pk=None):
    pass  # U - обновить (частично) по pk
  def destroy(self, request, pk=None):
    pass  # D - удалить по pk
```

ModelViewSet

ModelViewSet реализует все шесть методов — это удобно:
https://www.django-rest-framework.org/api-guide/viewsets/#modelviewset

```
class SomeViewSet(viewsets.ModelViewSet):
    queryset = SomeModel.objects.all()
    serializer_class = SomeSerializer
```

Как это работает

```
class SomeViewSet(viewsets.ModelViewSet):
  queryset = SomeModel.objects.all()
  serializer_class = SomeSerializer
```

- queryset нужен для поиска сущностей в БД
- serializer_class нужен для создания новых сущностей и изменения найденных сущностей с помощью queryset; **сериализатор** нужен не только для этого, это лишь одна из функций сериализатора

Роутер

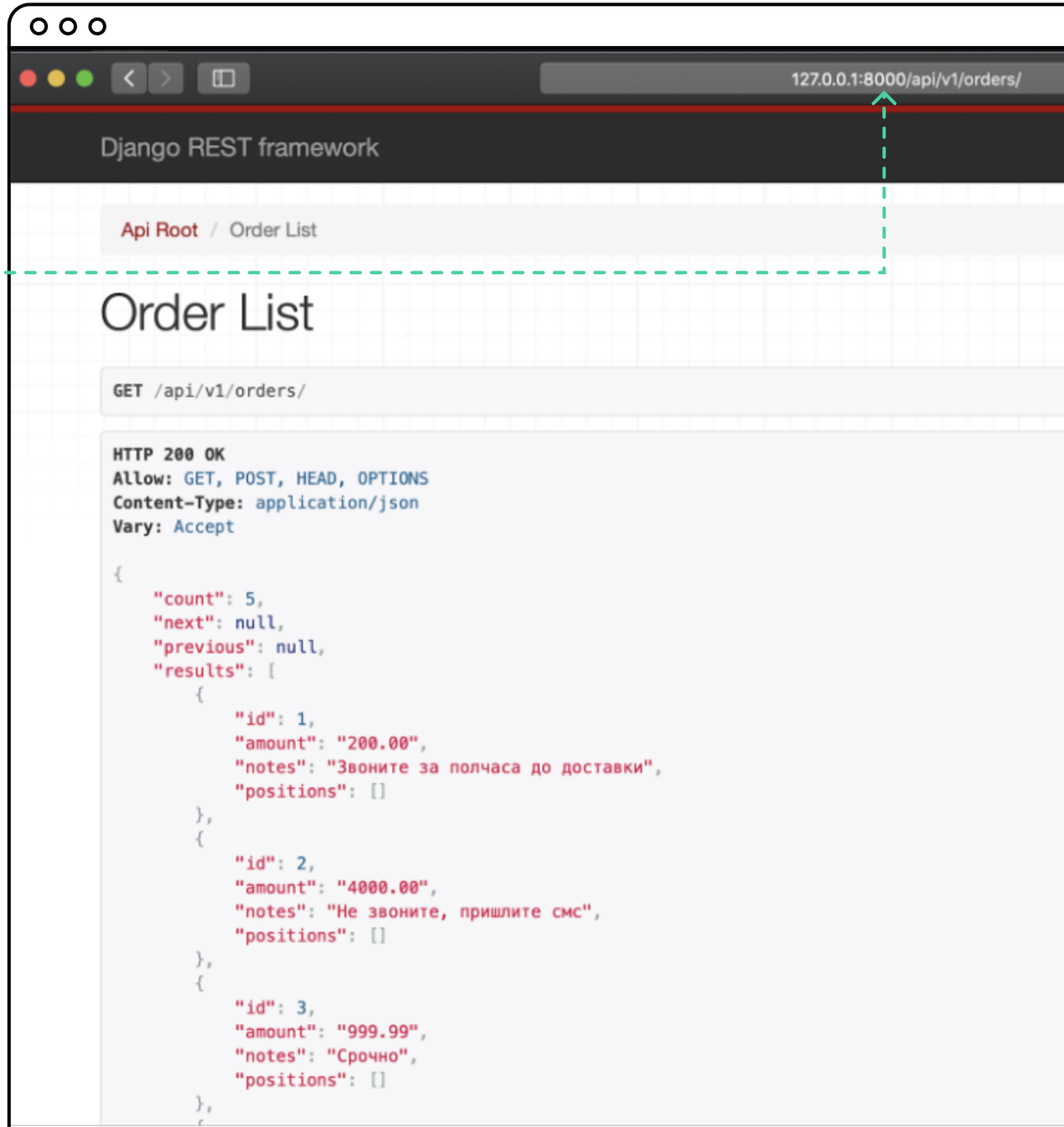
В роутере описываются **подключаемые view**. Роутер мапит HTTP-методы на методы view:

```
from django.urls import path, include
from rest_framework import routers

from orders import views as order_views

router = routers.DefaultRouter()
router.register('orders', order_views.OrderViewSet,
basename='orders')

urlpatterns = [

    path('api/v1/', include(router.urls)),

    path('admin/', admin.site.urls),

]
```

Результат

Специальная форма, которую генерирует DRF. Предназначена только для дебага.

В продакшене DRF будет возвращать JSON



Сериализаторы, валидация и обновление данных

Сериализаторы

Область ответственности сериализатора:

- представление данных
- валидация данных
- создание и обновление 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

Валидация данных

Валидация данных — это проверка данных на бэкенде. Нужно валидировать:

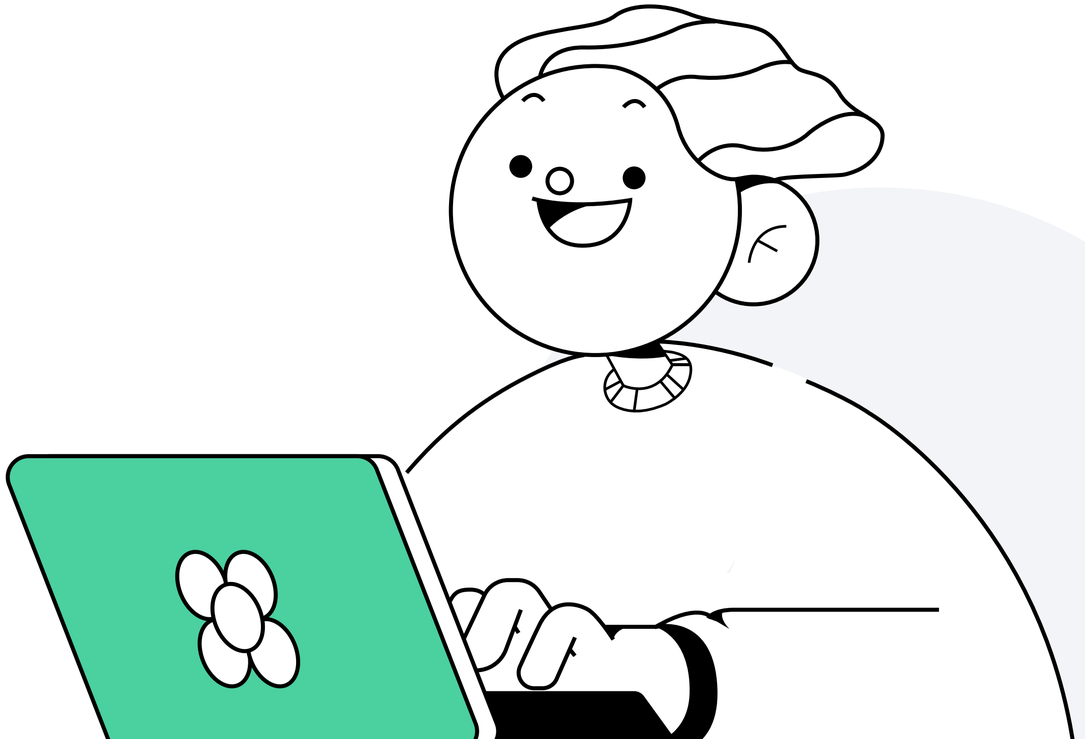
- **соответствие типов** — нельзя отправить строку вместо числа
- **специфичные ограничения** — нельзя создать товар или заказ с отрицательной стоимостью
- **бизнес-логику** — например, нельзя создать заказ без товаров

⚠ Никакие данные не должны записываться в базу данных без валидации

Валидация

Для валидации данных в DRF принято использовать сериализаторы.

Модель:



Код на следующей странице

Модели

```
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

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

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

Соглашения

В DRF принято описывать **сохранение данных в сериализаторе.**

Это похоже на приём, который используется в стандартных формах Django: https://docs.djangoproject.com/en/3.1/topics/forms/modelforms/

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

Соглашения

Валидация данных и сохранение/обновление **будут выполняться** автоматически, если вы используете **ModelViewSet**.

В противном случае вы можете сохранить модель **явно**, например, так:

```
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)
```

Сериализатор

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

```
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)
```

Сериализатор

Сериализатор должен **реализовывать методы** create и update, если хочется, чтобы он **умел создавать** и **обновлять** объекты (это не обязательно)
https://www.django-rest-framework.org/api-guide/serializers/#saving-instances

А вот ModelSerializer **уже** реализует эти методы. Это удобно.

Его надо только настроить:

```
class SomeSerializer(serializers.ModelSerializer):
  # тут может быть валидация
  class Meta:
    model = SomeModel
    fields = ['id', ...]
```

Сериализатор

Пример успешного ответа:

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

Фильтрация данных

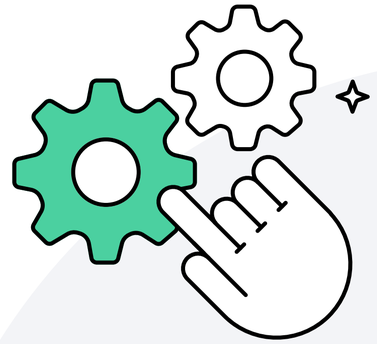
Фильтрация данных

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

Фильтры в DRF принято передавать через GET-параметры запроса:

Установка **django-filter:**

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



Фильтрация данных

Примечание: фильтрация применяется только для метода list (GET-запрос на получение списка).

Чтобы включить фильтрацию, необходимо либо задать DEFAULT_FILTER_BACKENDS в настройках, либо задавать filter_backends для каждого ViewSet:

```
from django_filters.rest_framework import DjangoFilterBackend

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

Разберём 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",)
```

Как ещё можно фильтровать

DRF предоставляет несколько возможностей **фильтрации и упорядочивания** результатов:

- DjangoFilterBackend — фильтрация по параметрам и значениям этих параметров
- SearchFilter — фильтрация-поиск, позволяет искать текст в указанных параметрах
- OrderingFilter — упорядочивание объектов по указанным параметрам

https://www.django-rest-framework.org/api-guide/filtering/#api-guide

Можно использовать сразу **несколько фильтраций:**

```
class OrderViewSet(viewsets.ModelViewSet):
    ...
    filter_backends = [DjangoFilterBackend, SearchFilter,
OrderingFilter]
    ...
```

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

Пагинация данных

Пагинация данных

Примечание: пагинация применяется только для метода list (GET-запрос на получение списка).

Чтобы **включить пагинацию**, необходимо либо задать DEFAULT_PAGINATION_CLASS в настройках, либо задавать pagination_class для каждого ViewSet:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 100
}
```

Другие варианты пагинаторов:

https://www.django-rest-framework.org/api-guide/pagination/#api-reference