# Django Rest Framework

Для начала проекта необходимо установить Django rest framework и создать проект, назовем его drf

pip install djangorestframework

# Set up a new project with a single application

django-admin startproject drf

Все манипуляции с Django происходят через файл управления manage.py, находящийся в корневой директории проекта

В файле настроек drf/settings.py пропишем настройки для drf:

REST\_FRAMEWORK = {

    'DEFAULT\_PERMISSION\_CLASSES': [

        'rest\_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'

    ],

    'DEFAULT\_PAGINATION\_CLASS': 'rest\_framework.pagination.PageNumberPagination',

    'PAGE\_SIZE': 10,

    'DEFAULT\_AUTHENTICATION\_CLASSES': [

        'rest\_framework.authentication.TokenAuthentication',

    ]

}

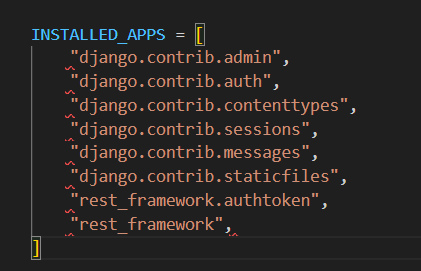
'DEFAULT\_PERMISSION\_CLASSES' – Устанавливаем правила доступа к моделям

'DEFAULT\_PAGINATION\_CLASS' – Определяем класс ответственный за пагинацию

'PAGE\_SIZE' – Определяем размер одной страницы

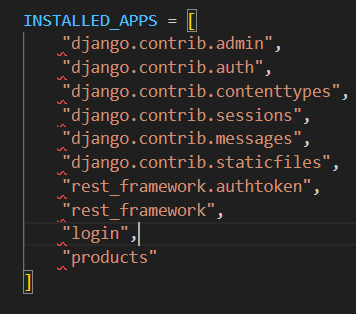
'DEFAULT\_AUTHENTICATION\_CLASSES – Указываем drf какие классы необходимо использовать для авторизации пользователя. 'rest\_framework.authentication.TokenAuthentication' позволяет пользователю авторизоваться с помощью токена

Также добавим restframework в список приложений



И сразу создадим 2 новых приложения login и products, отвечающие за авторизацию и товары соответственно

Их также регистрируем в настройках



После настройки сформируем и запустим миграции для создания нашей БД с помощью команды

python manage.py makemigrations

python manage.py migrate

Теперь создадим суперпользователя

python manage.py createsuperuser

Как альтернативный путь создания пользователя можно запустить проект и отправить запрос drf\requests\register.py

Первичная настройка закончена

Перейдем к функционалу пользователя.

В первую очередь создадим сераилизатор для модели пользователя (в данном проекте он нам не пригодится, но для общего понимания пусть будет) в файле login/serializer.py

class UserSerializer(serializers.ModelSerializer):

    """Сериализатор Пользователя для преобразования из json в объет python"""

    class Meta:

        model = User

        fields = "\_\_all\_\_"

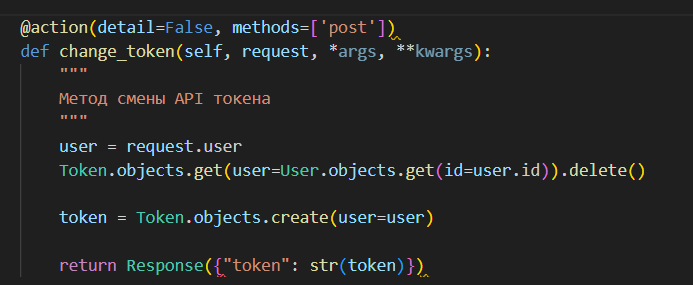
В данном случае мы используем ModelSerializer для упрощения кода. Его суть в том, что данный класс полностью опирается на модель User и сопоставляет поля для социализации с полями модели, поэтому дополнительно прописывать ничего не надо

Наследуемся от ModelSerializer и в подклассе Meta указываем модель на которую будет ссылаться сериализатор.

В поле fields указываем «\_\_all\_\_», что означает что сериализатор будет учитывать все поля из модели User.

Далее в файле drf\login\views.py создадим комплексное представление ModelViewSet

Его удобство в том, что у класс отвечает сразу за несколько методов взаимодействия с данными и нам не придется прописывать классы под каждое действие.



В данном классе нам понадобиться всего 1 метод для смены токена api. Создадим функцию change\_token, которая в качестве аргументов принимает self, объект запроса(request) и побочные аргументы.

Декоратор action указывает джанге на то, что эту функцию надо воспринимать как эндпоинт. Т.е. мы сможем обратиться к ней через параметр пути запроса (например <https://127.0.0.1/users/change_token> ) А также указываем, что данный экшн доступен только для post запроса

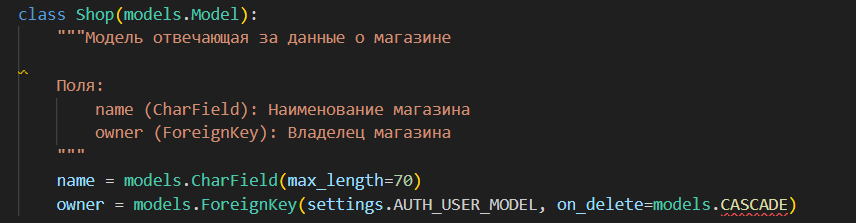
Далее дело за малым. Получаем пользователя из запроса и удаляем его токен, а затем создаем новый и возвращаем его пользователю через объект Response

Теперь перейдем к главному – маркетплейсу.

Начнем с описания моделей. Модель – Класс для работы с таблицей БД

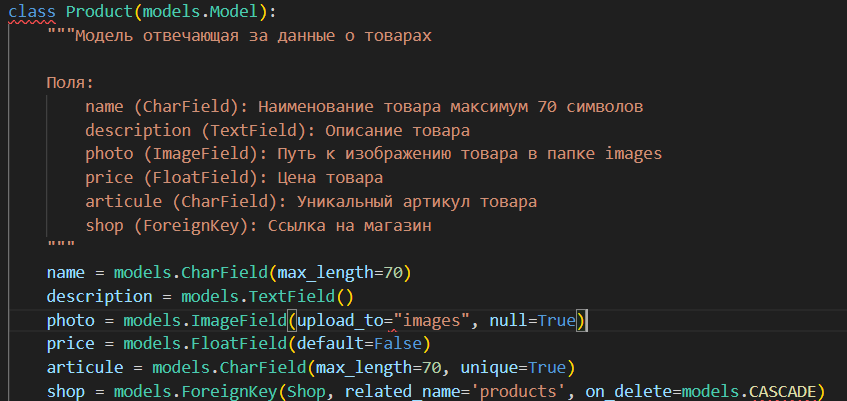
В файле drf\products\models.py напишем 2 модели Shop и Product

Модель shop отвечает за таблицу магазинов, а Product за товары.



В модели Shop всего 2 поля: Название и владелец. Все столбцы таблицы задаются классами полей модуля models. Так charfield предназначен для коротких строчек текста, а IntegetField для целочисленных значений.

Для столбца owner мы определили внешний ключ со ссылкой на модель пользователя



Для модели продуктов также определили поля. Стоит обратить внимание на столбец photo он задается классом ImageField, которое позволяет обрабатывать изображение на прямую и сохранять их в папку, указанную параметром upload\_to. Null=True говорит джанге, что это поле может содержать None

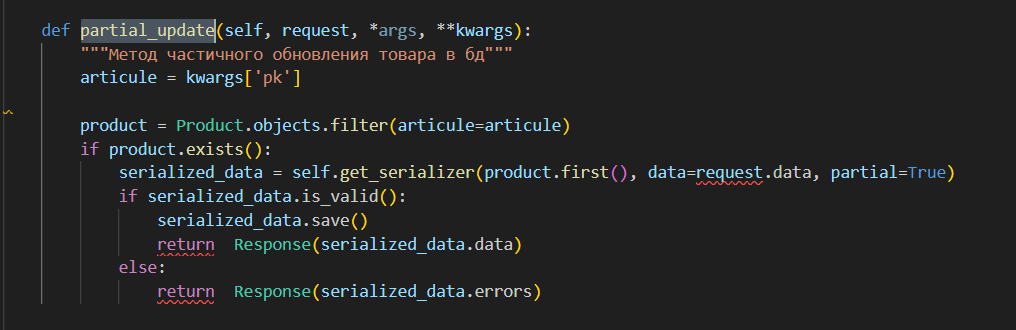
А в столбце shop параметр related\_name указывает на обратную связь от объекта Shop к этой таблице. Т.е. мы можем обратиться к products как shop\_obj.products

После написания моделей создадим и применим миграции снова. Как это сделать показывалось выше.

Далее создаем ViewSet для нашего маркетплейса в файле drf\products\views.py

В этом viewSet’е у нас есть дополнительные методы retrieve, create, destroy, partial\_update. Это такие обработчики запросов GET, POST, DElETE и PATCH соответственно. Определяя их, мы можем настраивать поведение api при этих методах.

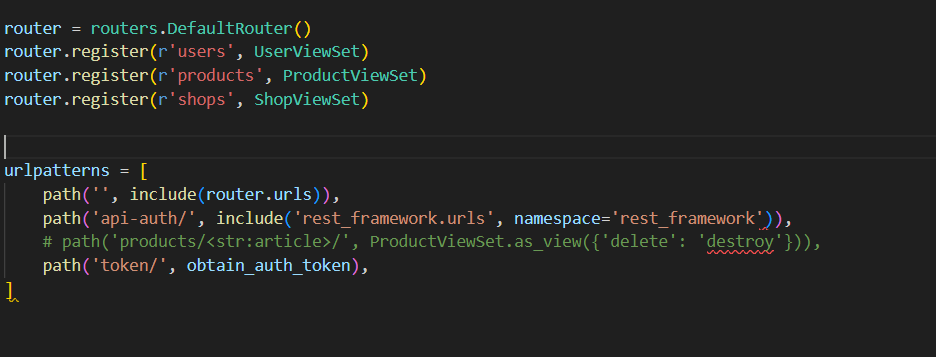
В качестве kwargs аргументов в них попадают path параметры, например в partial\_update мы передаем артикул в параметре пути (<https://127.0.0.1/products/13141/> 13141 это и есть наш артикул)



С помощью полученного артикула получаем объект товара и проверяем его на существование. Далее указываем нашему сериализатору, какой объект мы хотим модифицировать, передаем данные, полученные из запроса с помощью request.data, а также указываем, partial=True ведь мы хотим лишь частично обновить объект.

После валидируем данные. Если все ок, сохраняем и отправляем ответ пользователю

Остальные методы похожи на этот. Отличаются они не сильно только конечным действием: Созданием/Удалением/Редактированием объекта

Не забываем зарегистрировать наши представления в url.py 

Чтобы Джанга могла распарсить строку запроса и правильно подставить нужный роут.

В папке drf\requests лежат скрипты для тестирования написанные на requsts

# FastAPI

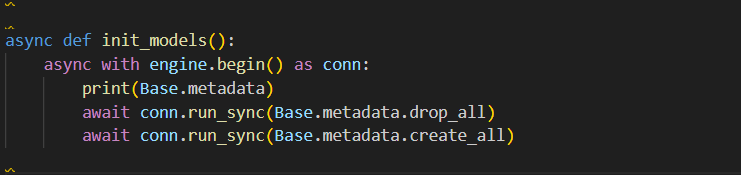
Первым делом установим зависимости

pip install -r requirements.txt

Далее при необходимости можно сформировать новую БД, расскоментировав строчку в файле main.py

    asyncio.run(init\_models())

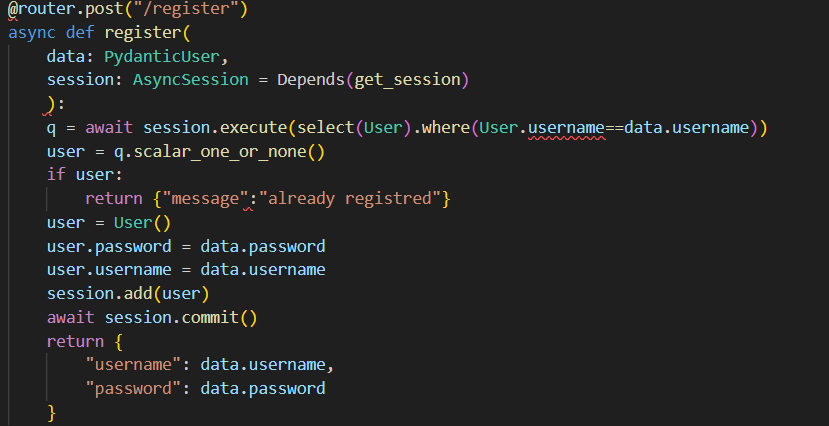
эта функция вызывает сброс и создание всех таблиц базовой модели



В главном файле также подключены роутеры, отвечающие за определение пути. В принципе можно было оставить все хэндлеры в главном файле, но в угоду красоты кода вынес их в отдельные роуты.

Переходим непосредственно к хэндлерам

Первый обработчик — это регистрация пользователя

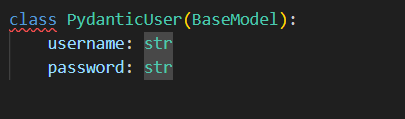


Суть проста. Получаем имя пользователя и пароль, если такого пользователя еще нет, создаем запись в бд. В большом проекте открытое хранение пароля недопустимо по причинам безопасности. Но в данном случае опустим это.

Стоит обратить внимание на аргументы функции.

Первый аргумент это Pydantic класс. Pydantic – библиотека отвечающая за социализацию данных получаемых от пользователя в fastapi

Класс PydanticUser, как и другие Pydantic классы определены в файле fastapi\serializer.py



Указывая этот класс как аннотацию к переменной form\_data, мы говорим fastapi валидировать входящие данные по этому классу. В данном случае в функцию будет приниматься только json объект, в котором 2 поля «username» и «password» и оба строки. Иначе мы получим ошибку валидации.

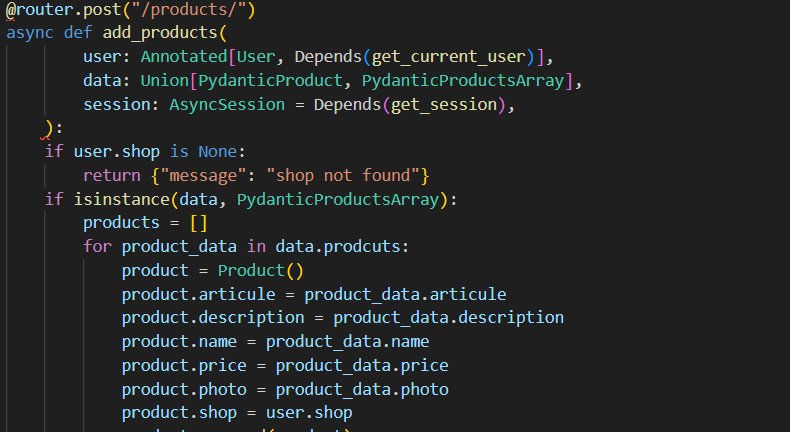
Вторым аргументом передается сессия. В FastAPI реализована система зависимостей, которая позволяет вызывать необходимые для данного хэндлера методы в иерархической последовательности.

В данном случае через зависимость

Depends(get\_session)

Мы получаем объект сессии sqlalchemy для работы с БД

Далее рассмотрим обработчик добавления товаров



По сути, все тоже самое, отличается только логика работы. Единственное отличие — это получаемый от клиента запрос. Здесь мы используем Union для того, чтобы fastAPI понял, каких данных следует ожидать. Если придет PydanticProductsArray, то будет выполнено множественное добавление записей. В противном случае будет добавлена только одна запись.

Общая схема работы с api такая:

1. Регистрация. Файл fastapi\requests\register.py
2. Получение токена. fastapi\requests\token.py
3. Создание магазина. fastapi\requests\create\_shop.py
4. Работа с товарами:
   1. Создание fastapi\requests\create.py
   2. Удаление fastapi\requests\delete.py
   3. Получение fastapi\requests\get.py
   4. Обновление fastapi\requests\update.py