МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Челябинский государственный университет» (ФГБОУ ВО «ЧелГУ»)

Колледж ЧелГУ

09.02.04 Информационные системы (по отраслям) КУРСОВОЙ ПРОЕКТ

По дисциплине Web-программирование На тему:

**Создание Web-сайта для генерации рассписания**

Выполнил: студент СПМИС-301

Киселев Артемий Кириллович Руководитель:

Кирилл Лихачев

Оценка

Подпись

Миасс

2023

СОДЕРЖАНИЕ

|  |  |
| --- | --- |
| ВВЕДЕНИЕ……………………………………………………......... | 3 |
| 1 Теоретическая часть....……………………………………..... | 5 |
| 1.1 Анализ требований.................................................................. | 5 |
| 1.2 Выбор технологического стека............................................... | 6 |
| 1.3 Проектирование интерфейса и архитектуры приложения.... | 8 |
| 2 Разработка серверной части (Backend).................................. | 14 |
| 2.1 Реализация базы данных.......................................................... | 14 |
| 2.2 Создание проекта и разработка API....................................... | 19 |
| 2.3 Разработка алгоритма генерации рассписания..................... | 22 |
| 2.4 Использование системы контроля версий git............................. | 22 |
| 3 Разработка клиентской части (Frontend)............................... | 14 |
| 3.1 Реализация пользовательского интерфейса.......................... | 14 |
| 3.2 Интеграция с серверной частью............................................. | 14 |
| 3.3 Отображение результата......................................................... | 14 |
| 4 Развертывание приложения................................................... | 14 |
| 4.1 Подготовка nginx сервера для развертки приложения............... | 14 |
| 4.2 Развертывание на удаленном сервере.......................................... | 14 |
| 4.3 Инструкция к использованию приложения................................. | 14 |
| ЗАКЛЮЧЕНИЕ.................................................................................. | 26 |
| СПИСОК ИСПОЛЬЗОВАНЫХ ИСТОЧНИКОВ............................ | 27 |

**ВВЕДЕНИЕ**

В эпоху цифровой трансформации, когда даже самые обычные задачи становятся автоматизированными, появляется все больше потребностей в различных веб-сервисах. Один из таких сервисов - это веб-сайт для генерации расписания. Будь то расписание для школьников, студентов, сотрудников компании или даже тренажерного зала, цифровой подход значительно упрощает процесс управления временем и ресурсами.

Создание веб-сайта для генерации расписания – это многоуровневый процесс, который включает в себя понимание требований пользователей, проектирование интерфейса, разработку логики функционирования и поддержку в рабочем состоянии. Для эффективной реализации проекта необходимо не только технические навыки, но и глубокое понимание бизнес-процессов, к которым прикладывается система.

В рамках этой задачи мы подробно рассмотрим все аспекты создания веб-сервиса для генерации расписания, начиная от анализа требований и заканчивая этапами внедрения и поддержки. Мы исследуем важные темы, такие как выбор технологического стека, определение функциональности, разработка дизайна и интерфейса.

Целью данного изложения является предоставление полного и понятного представления о процессе создания веб-сайта для генерации расписания, чтобы помочь начинающим разработчикам и другим заинтересованным лицам успешно реализовать подобные проекты.

1. **Теоретическая часть**
   1. **Анализ требований**

Сервис должен быть способен обрабатывать данные о различных группах и соответствующем количестве пар в неделю. Это подразумевает необходимость разработки алгоритма, способного оптимально распределить занятия на протяжении недели, с учетом того, что преподаватель может вести несколько предметов в разных группах, а также предметы могут занимать одинаковые кабинеты.

Важной функциональностью сервиса будет возможность добавления, удаления и редактирования групп, предметов, учителей и кабинетов их нагрузки в базе данных, что обеспечит гибкость и актуальность информации. Также, сервис должен обеспечивать генерацию расписания в удобном для пользователя формате, с возможностью его просмотра на сайте.

Основываясь на этих данных, можно определить первоначальные требования к функциональности сайта, которые послужат основой для дальнейшего проектирования и разработки.

* 1. **Выбор технологического стека**

В качестве технологического стека для серверной части мы можем использовать несколько вариантов:

1. Flask
2. Fast Api
3. Django
4. GoLang RPC

Рассмотрим их поподробнее:

1. Flask – довольно простой фреймворк, использующий шаблонизатор Jinja 2 для рендеринга html страниц.   
   Плюсы: Достаточно простой для изучения, подходит для небольших приложений

Минусы: Очень мало функций из коробки, для установки дополнительных потребуется их поискать, и почитать что они делают. Нет встроенной ORM, придется также подключать SQLAlchemy и самостоятельно подключаться к Базе данных.

1. Fast Api – асинхронный фреймворк для написания API серверного приложения.

Плюсы: Асинхронный, подходит для написания быстрого API для небольших или даже высоконагруженных систем.

Минусы: Нет поддержки рендеринга страниц, что означает нам придется использовать JavaScript для динамической отрисовки страницы по данным с сервера. Нет ORM из коробки, также придется подключать SQLAlchemy и самостоятельно делать подключения к базе данных.

1. Django – отличный веб фреймворк для написания веб приложений любой сложности. Поддерживает как синхронную, так и ассинхронную версию

Плюсы: Легко настроить и подключить приложения, есть хорошая библиотека DRF (Django REST Framework) для написания API любой сложности. Есть встроенная ORM и отрисовщик шаблонов. Отрисовщик шаблонов позволяет принимать даже ORM модели. Легко пишутся модели для базы данных

Минусы: Довольно много функций из коробки, много функций может не понадобиться. Порог вхождения выше, чем у предыдущих фреймворков.

1. GolangRPC – библиотека, написанная на языке программирования Go. Позволяет сделать быстрый ассинхронный простой веб-сервер. Сам язык Go очень минимизирован и построен на архитектуре языка C, а также выполняется ассинхронно.

Плюсы: Идеально подходит для написания микросервисной архитектуры. Также идеально подходит для написания совсем небольшого API приложения. Выполняет ассинхронные запросы в Базу Данных в рамках одной транзакции, что ускоряет различные запросы в БД.

Минусы: не поддерживает рендер html документов из коробки, есть пользовательская, но очень неудобная в использовании ORM (gorm). Требуется понимание ассинхронной архитектуры приложения и самого устройства языка программирования Go.

На основе этой информации я решил выбрать связку Django + Django REST Framework. Оно идеально подходит под задачу: можно легко сделать модели кабинетов, групп, учителей и предметов в базу данных, легко фильтровать модели в базе данных. Также можно сделать как полноценный веб-сервер с динамическими html страницами за счет рендера, встроенного в сам фреймворк, так и для написания удобного API с готовой сериализацией и методами для настройки маршрутизации.

В качестве алгоритма генерации сначала я хотел использовать микросервис, написанный на gRPC, но в последствии отказался от этой идеи, и добавил к Django приложению брокер сообщений redis и очередь задач Celery, на которой и будет

размещен алгоритм генерации. Они также написаны на Python, не придется использовать 2 разных языка и переносить модели в другой язык.

Также нам потребуется Docker, для объединения всех этих технологий в одну сеть из изолированных контейнеров, где они гарантированно запустятся, вне зависимости от операционной системы пользователя.

Для Frontend части приложения мы можем использовать:

1. Vue.js – Довольно простой JS фреймворк для построения простых клиентских приложений
2. React.js – Хороший JS фреймворк, поддерживающий типизацию на TypeScript, а также имеющий много полезных библиотек к нему

В качестве Frontend фреймворка я выбрал React, так как у меня уже есть опыт работы с ним, и мне будет проще работать на нем.

И последнее, для соединения Frontend и Backend частей мы будем использовать nginx сервер, который будет поддерживать оба этих приложения.

* 1. **Проектирование интерфейса и архитектуры приложения**

Основная задача нашего приложения - генерация расписания. Для этого нам понадобится 4 сущности в базе данных:

1. Группы, у которых есть связи с парами
2. Пары, у которых есть привязанный кабинет и преподаватель
3. Преподаватели
4. Кабинеты

Создадим 4 модели в нашем Backend приложении:

1. Модель предмета



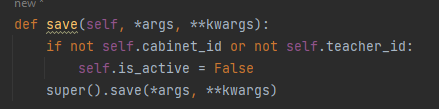
У этой модели есть 4 поля:  
name – отображаемое название для предмета, которое показывается пользователю

is\_active – индикатор того, что пара может быть использована для группы и генерации расписания

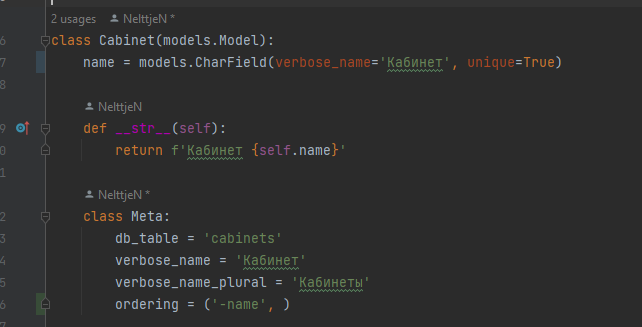
cabinet – Связь с моделью кабинета, может быть пустой, при удалении поле становится null

teacher – Связь с моделью преподавателя, может быть пустой, при удалении поле становится null

Также нам нужно перегрузить метод save класса models.Model, чтобы если к предмету не был привязан учитель или кабинет – он не считался активным, и не мог использоваться для генерации расписания:



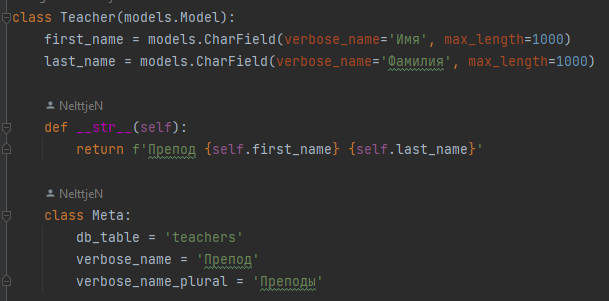
1. Модель кабинета



Здесь у нас есть только поле name для отображения названия кабинета. Например 301 или сп.зал. Название должно быть уникальным, т.к. у нас не может быть несколько одинаковых кабинетов в колледже.

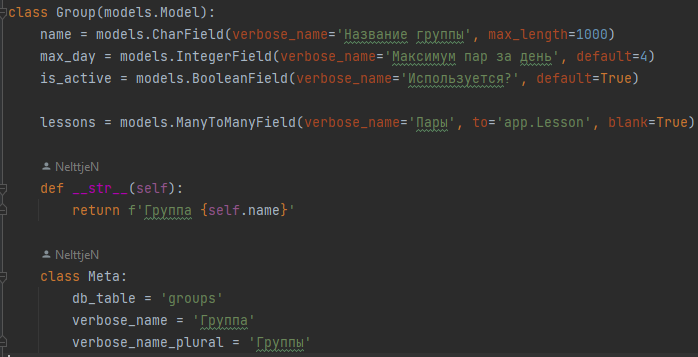
Также в подклассе Meta мы определяем поле ordering, чтобы кабинеты располагались в порядке возрастания

1. Модель Преподавателя:



У этой модели есть Имя и Фамилия преподавателя.

1. Модель группы:



У этой модели есть поля:

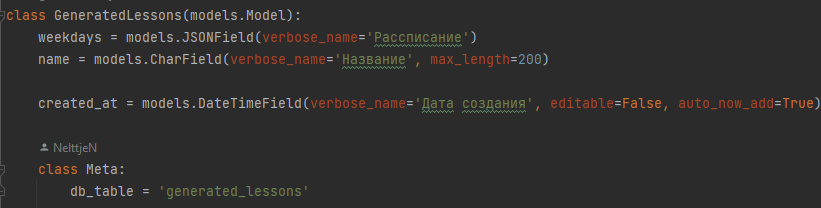
name – отображаемое название группы (например можно вписать мою группу СПМИС-301, это и будет название группы).

max\_day – количество пар, которое алгоритм может максимум поставить в день. По умолчанию – 4.

is\_active – отображается ли группа в списке доступных групп для алгоритма генерации.

lessons – связь с моделями предметов, те что выбраны – будут отображаться в списке для генератора рассписания.

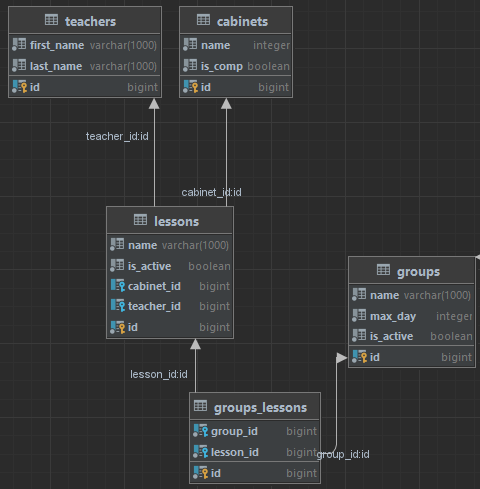
Также нам понадобится ещё одна модель – для уже сгенерированных расписаний:



name – отображаемое название расписания, которое можно выбрать на стадии настройки генерации

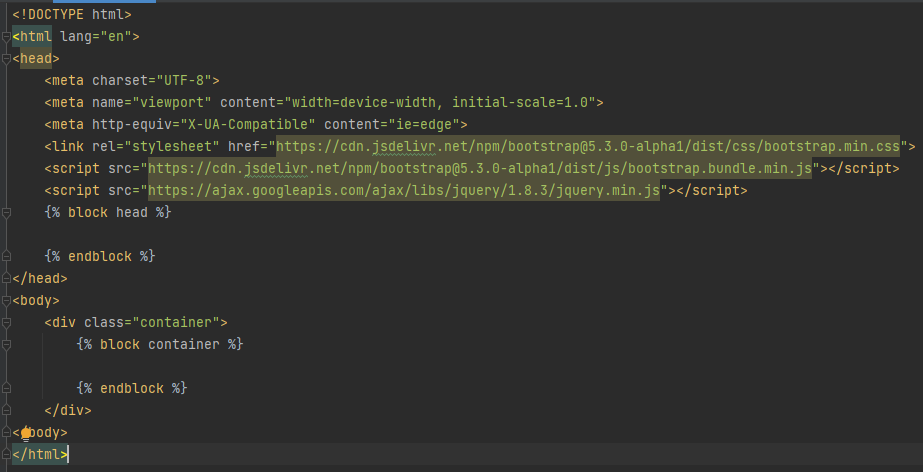
weekdays – json поле, в котором будет храниться json объект сгенерированного расписания, который будет использоваться Frontend приложением для отображения рассписания.

created\_at – дата создания, для сортировки и отображения уже сгенерированных расписаний.

Визуализация всех связей и моделей в базе данных:  


Интерфейс приложения

Для создания интерфейса нам понадобится создавать динамические html страницы, для отображения контента из базы данных. Поэтому создадим базовый html шаблон, от которого будем наследовать все остальные



В нем мы создадим 2 блока: head и container. В блок head мы будем вставлять всё, что будет вставляться в html блок <head> дополнительно от базовых библиотек, в container будет вставляться основная страница.

Также в head определены базовые библиотеки, которые будут использоваться:  
CSS Bootstrap 5, JS Bootstrap 5 и JQuery. Они используются для упрощения некоторых методов в JS скриптах и использовании готовых вариантов дизайна разных элементов (кнопки, карточки, формы и т.д.)

Интерфейс настроек генератора будет сделан на обычном шаблонизаторе Django, а интерфейс сгенерированных расписаний будет на React App.

1. **Разработка серверной части (Backend)**
   1. **Реализация базы данных**

В качестве базы данных мы можем использовать несколько реляционных баз данных, таких как:

1. PostgreSQL
2. MySQL
3. SQLite

Рассмотрим их:

1. PostgreSQL – Хорошая реляционная база данных, имеющая множество преимуществ из коробки в области организации, репликации и оптимизации хранения данных, а также довольно простым SQL синтаксисом. Подойдет для проектов любого масштаба.
2. MySQL – Проверенная временем реляционная база данных, с немного отличным от привычного SQL синтаксисом (совсем чуть-чуть). Подойдет для проектов любого масштаба, но при масштабировании могут возникнуть некие сложности, которые не возникнут в PostgreSQL
3. SQLite – Маленькая база данных, хранящаяся в одном файле. Подойдет для маленьких проектов, не нужно поднимать даже сервера, т.к. подключение идет напрямую из фреймворка

Для проекта я выбрал базу данных PostgreSQL, т.к. для меня у неё более привычный SQL синтаксис, а также есть отличный инструмент для мониторинга базы данных – PgAdmin 4

Чтобы развернуть сервер базы данных я использую готовый официальный контейнер PostgreSQL, доступный на Docker hub: <https://hub.docker.com/_/postgres>, и выберу версию 15.3-alpine. 15.3 самая новая версия на момент написания курсовой работы, а приставка alpine означает что эта версия идет как самый минимальный набор операционной системы, необходимый для её поддержания. В рамках контейнера, когда каждый контейнер представляет из себя некую «виртуальную» машину с операционной системой на ядре Linux, это ускорит запуск и работу самой базы данных, так как система не будет отвлекаться на прочие ненужные модули, а будет загружать только необходимое. Приступим к созданию базы данных.

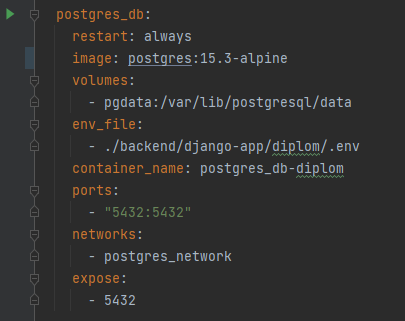
Для начала в корне нашего приложения создадим пустой файл docker-compose.yaml. docker-compose – технология, позволяющая легко объединять несколько контейнеров в одну сеть, а также использовать готовые Dockerfile с docker hub. Что мы как раз и делаем, указывая версию для PostgreSQL:15.3-alpine. Docker-compose автоматически скачает нужный Dockerfile для заданной версии и соберет из него контейнер, или же как они называются в самом Docker – image.

В самом верху указываем версию docker-compose. У меня это будет 3.7.



Далее добавляем блок services, это и есть блок, в котором описаны все контейнеры, которые будут собраны командой docker-compose build и запущены командой docker-compose up.

В нем мы добавляем сервис postgres\_db и указываем нужные поля:



restart – отвечает за перезапуск запущенного контейнера. В моем случае always означает что даже если произойдет ошибка, он будет перезапускаться до тех пор, пока я сам не выключу контейнер.

image – отвечает за получение Dockerfile с Docker Hub и последующем его использовании для построения данного контейнера.

volumes – отвечает за связь папки на хосте с папкой внутри контейнера. В данном случае pgdata это отдельно созданный volume, в котором хранится все файлы нашей базы данных, чтобы при пересборке контейнера у нас не пропали все

сохраненные в нем данные.

env\_file – отвечает за путь к файлу со значениями переменной среды на хосте. Альтернативно можно использовать ключевое слово environment, но данные для подключения к базе данных нам пригодятся не только в этом контейнере, но ещё и в нашем Django приложении. И чтобы не дублировать их, и при изменении не лезть в несколько мест и используется один файл для задания этих настроек. Обычно такой файл называется .env или .environment

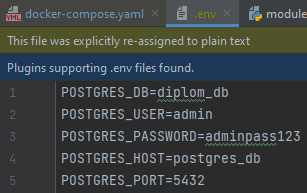
container\_name – отвечает за отображаемое название контейнера. Это нужно, чтобы предотвратить «коллизию» нескольких контейнеров. Допустим у нас нужно на одном сервере поставить 4 базы данных, у каждой из которых отдельная директория и отдельный docker-compose файл. По умолчанию docker-compose будет брать ключевое слово сервиса, а у нас будет ключевое слово postgres\_db во всех 4 файлах. 1 файл соберется и успешно запуститься, но остальные не смогут, так как такой контейнер уже существует.

ports – отвечает за соединения порта внутри контейнера с портом на хосте. Для всех локальных контейнеров будет порт, указанный вторым числом. Например мы выставим значение “443:5432”. Если мы хотим сделать локальное подключение, например из другого контейнера, для него подключение будет доступно по адресу postgres\_db:5432. Но если мы захотим подключиться к базе данных с другого сервера, или с отдельного приложения, которое не запущено в контейнере, нам придется использовать порт 443, т.к. порт 5432 существует только внутри изолированной среды контейнеров, и не взаимодействует с внешним окружением.

networks – служит для связи нескольких контейнеров в одну локальную сеть, для примера как связать несколько компьютеров через роутер, только тут несколько контейнеров через сеть. Изначально все контейнеры, у которых не определен этот параметр связаны через одну общую сеть. Но например мы не хотим, чтобы какое-то наше приложение имело связь с другим приложением, и эти 2 приложения будут связаны в 2 разных сети, и не будут видеть друг-друга локально.

expose – служит для пробрасывания портов только внутри локальных контейнеров, но не наружу. Таким образом доступ к этому порту будет только внутри локальной сети, но не будет доступа к этому порту снаружи контейнера.

В качестве переменных окружения в .env файле нам нужно указать следующие переменные:

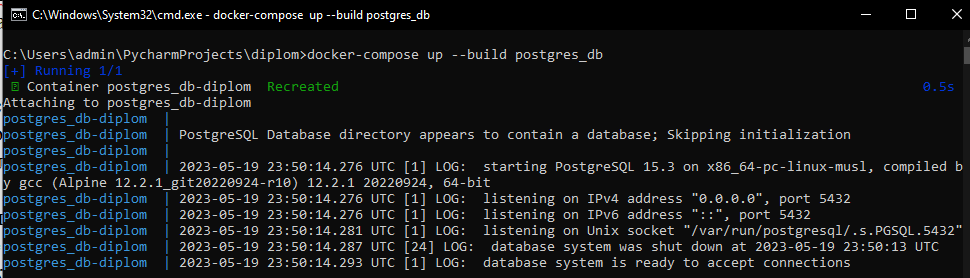


Здесь мы указываем что наш проект будет использовать базу данных с названием diplom\_db, а также создаем в базе данных пользователя пользователя admin с паролем adminpass123

В качестве хоста и порта мы указываем название контейнера и порт 5432 (стандартный порт PostgreSQL). Так как наш проект будет развернут в контейнере, чтобы подключиться к базе данных можно использовать название контейнера в качестве хоста и порт.

У каждого контейнера есть свой локальный ip адрес, к примеру, у postgres\_db будет адрес 127.0.0.24, но иногда он может меняться, если мы изменили сеть или добавили ещё контейнеров, или же просто пересобрали их. В таком случае, чтобы постоянно не узнавать новый ip контейнера, используется его название, указанное в блоке services. Docker автоматически найдет контейнер и поставит его ip для подключения. В таком случае <http://postgres_db:5432> будет эквивалентно <http://127.0.0.24:5432> внутри контейнера.

После этого открываем командную строку и запускаем контейнер с базой данных коммандой docker-compose up --build postgres\_db



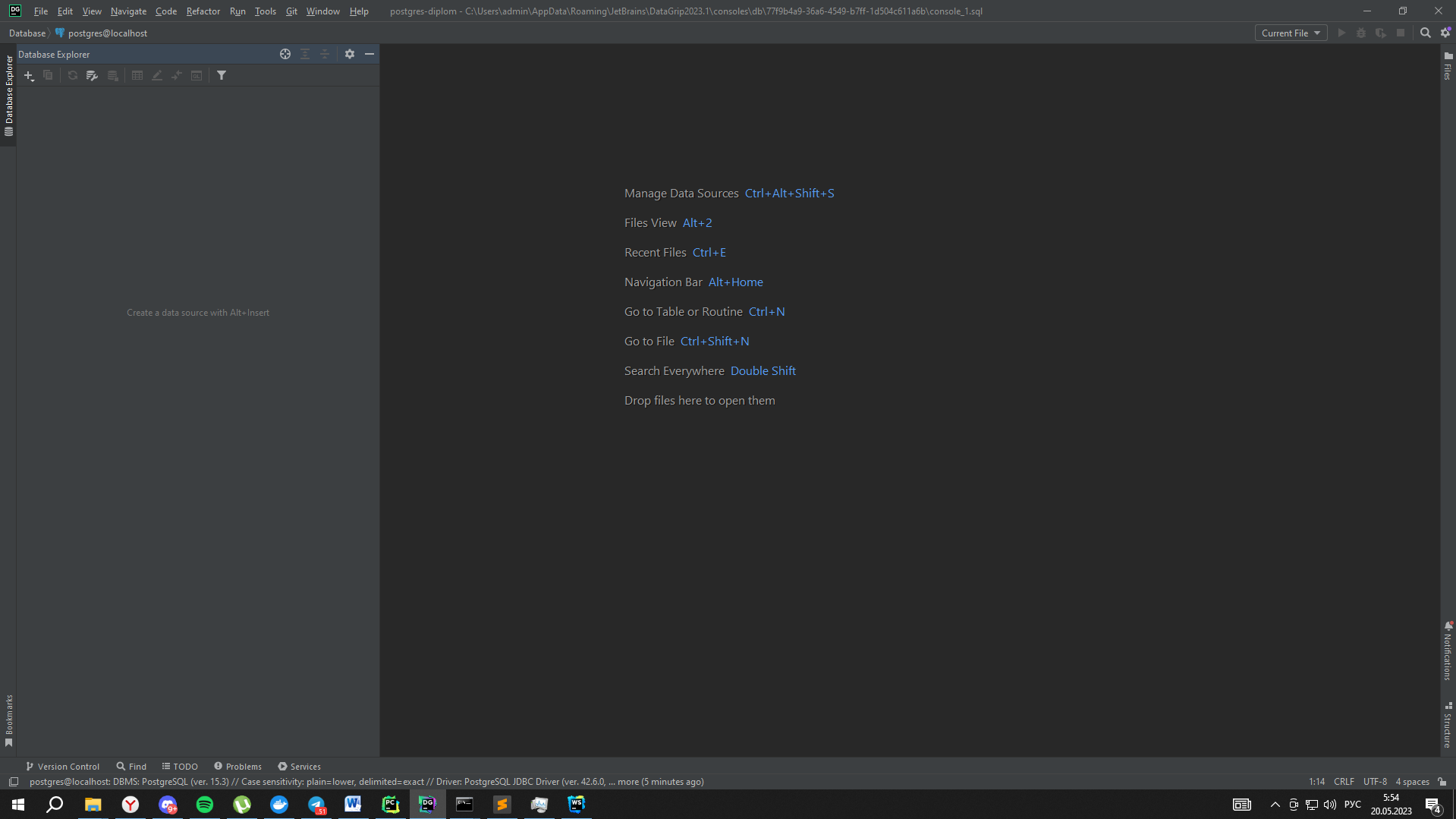
Как мы видим, база данных успешно развернулась на локальном порту внутри контейнера 5432. Так как в docker-compose мы указывали связать порт 5432 на хосте

с портом 5432 внутри контейнера, а хост является нашим компьютером, то база данных должна быть доступна по адресу: <http://127.0.0.1:5432> или <http://localhost:5432>.

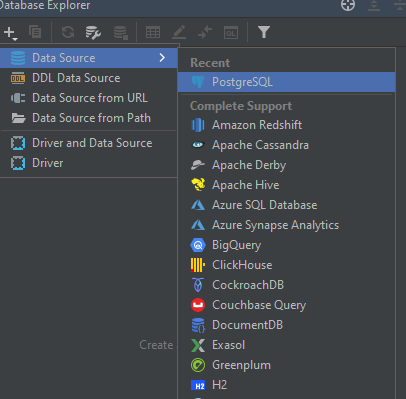
Чтобы это проверить, мы открываем DataGrip. DataGrip - это интегрированная среда разработки (IDE) для работы с базами данных, разработанная компанией JetBrains. Она предоставляет мощный и удобный инструментарий для работы с различными типами баз данных, включая SQL, PostgreSQL, MySQL, Oracle, Microsoft SQL Server и другие.

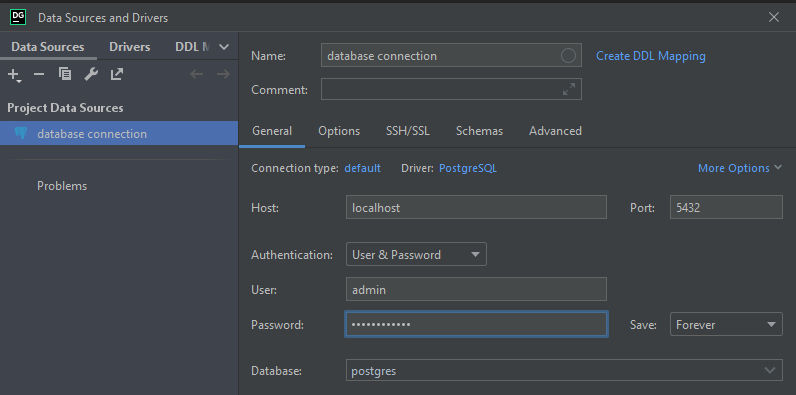
DataGrip обладает широким спектром функций, которые облегчают разработку, администрирование и анализ данных. Одной из ключевых особенностей DataGrip является его удобный и интуитивно понятный интерфейс. Он предоставляет удобные инструменты для работы с базами данных, такие как подсветка синтаксиса SQL, автодополнение кода, навигация по объектам базы данных и многое другое. Это позволяет разработчикам более эффективно работать с базами данных и сократить время, затрачиваемое на написание и отладку SQL-запросов.

DataGrip также предоставляет множество функций для администрирования баз данных. Это включает в себя возможность просмотра и изменения данных, создание и редактирование таблиц, управление пользователями и правами доступа, выполнение массовых операций обновления и многое другое. Благодаря своей интеграции с различными СУБД и расширенным функционалом, DataGrip позволяет разработчикам эффективно работать с базами данных и повышать производительность своих проектов.

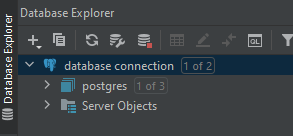
В DataGrip мы создаем новый проект, и у нас откроется его интерфейс:

Здесь мы видим кнопку добавить новую базу данных, в открывшемся окне выбираем что мы хотим подключиться к PostgreSQL:

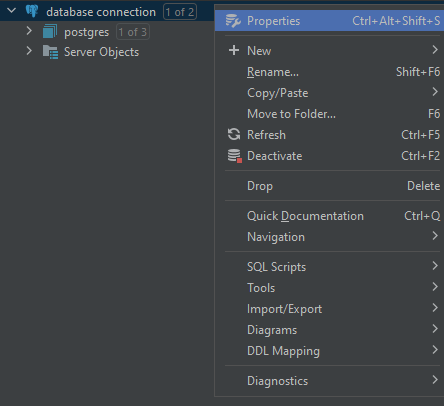


В открывшемся окне мы можем задать название подключения, я назову “database connection”. В качестве параметра host вставим 127.0.0.1, что означает что хост – наш локальный компьютер. Порт 5432, как и указывали в docker-compose. Авторизацию выбираем по пользователю и паролю, и так как в .env у нас указан пользователь admin с паролем adminpass123, это и есть данные для авторизации в базу данных. Окно с добавлением подключения должно теперь выглядеть так:  


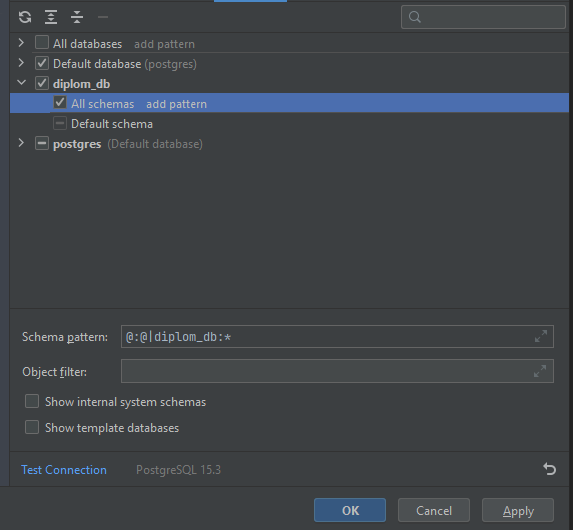
Нажимаем кнопку принять и если мы всё сделали правильно – у нас должно появиться активное подключение



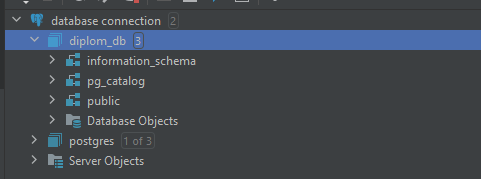
Но в данном случае мы видим только базу данных postgres, которая является служебной базой данных сервера PostgreSQL, и нам не интересна. Чтобы увидеть нашу базу данных diplom\_db, кликаем пкм по database connection и выбираем пункт параметры (или properties, так как у меня установлена английская версия)



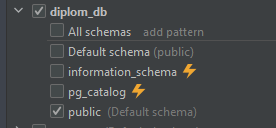
В открывшемся окне выбираем Schemas, видим базу данных diplom\_db, разворачиваем и ставим галочку у параметра «Все схемы»



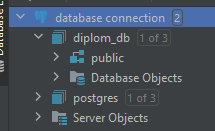
Нажимаем принять и смотрим результат.



У нас появилась база данных diplom\_db и 3 схемы: information\_schema, pg\_catalog и public. Первые 2 нам не интересны, а в public хранятся все наши данные и созданные таблицы. Давайте отключим первые 2 схемы, чтобы они не мешали нам в том же окне Параметры > Схемы > diplom\_db:



Нажимаем применить, смотрим результат:



Всё как нужно, не интересующие нас схемы нам больше не показываются.

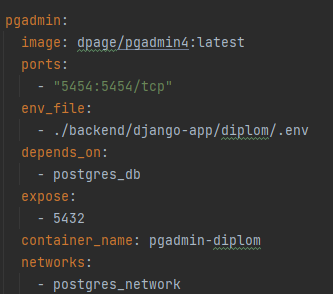
Схема information\_schema предоставляет метаданные о базе данных и ее объектах. Она содержит системные представления, которые предоставляют информацию о таблицах, столбцах, ограничениях, индексах и других объектах в базе данных. Её можно использовать для получения доступа к информации о структуре базы данных и использовать ее для различных целей, таких как создание отчетов, проверка целостности данных, анализ схемы и другие административные задачи.

Схема pg\_catalog является системной схемой, которая содержит метаданные и определения объектов, специфичных для PostgreSQL. Она предоставляет доступ к системным представлениям и функциям, которые описывают внутреннее состояние базы данных и ее компонентов. Схема pg\_catalog содержит информацию о системных таблицах, типах данных, операторах, функциях и других объектах, которые являются основой работы PostgreSQL. Её можно использовать для получения информации о внутреннем состоянии базы данных, выполнения анализа запросов, оптимизации производительности и выполнения других задач, связанных с управлением и настройкой PostgreSQL.

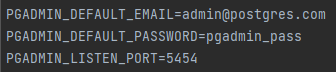
На этом настройка базы данных окончена. Наша база данных готова к работе с проектом.

Для удобства я также планирую развернуть сервис PgAdmin 4, чтобы можно было мониторить существующие подключения к базе данных, смотреть её состояние и данные, хранящиеся в ней. Для этого я беру готовый вариант PgAdmin 4

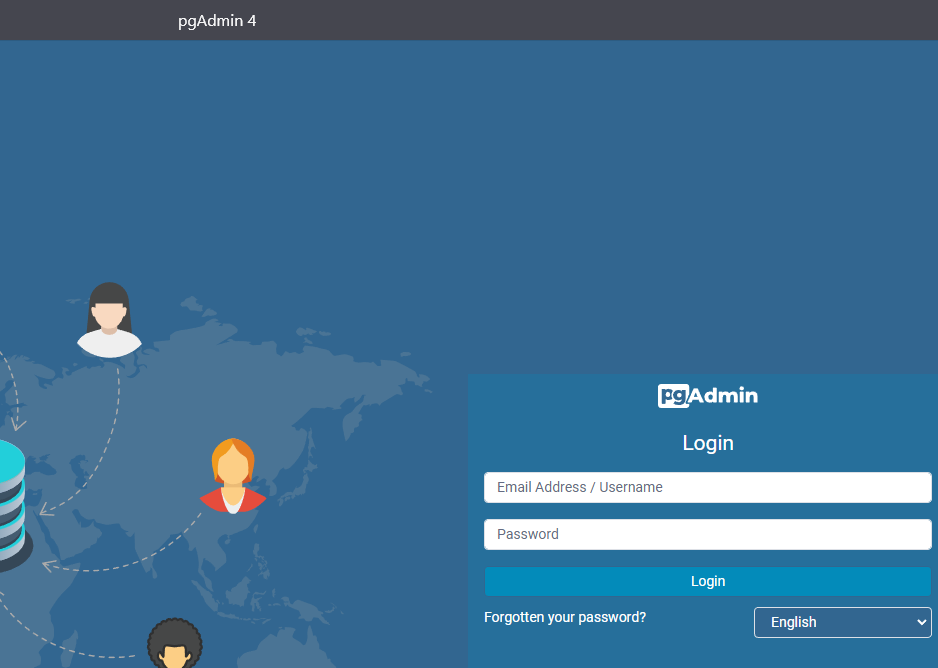
найденый на Docker Hub <https://registry.hub.docker.com/r/dpage/pgadmin4> и создаю для него сервис в нашем docker-compose файле:



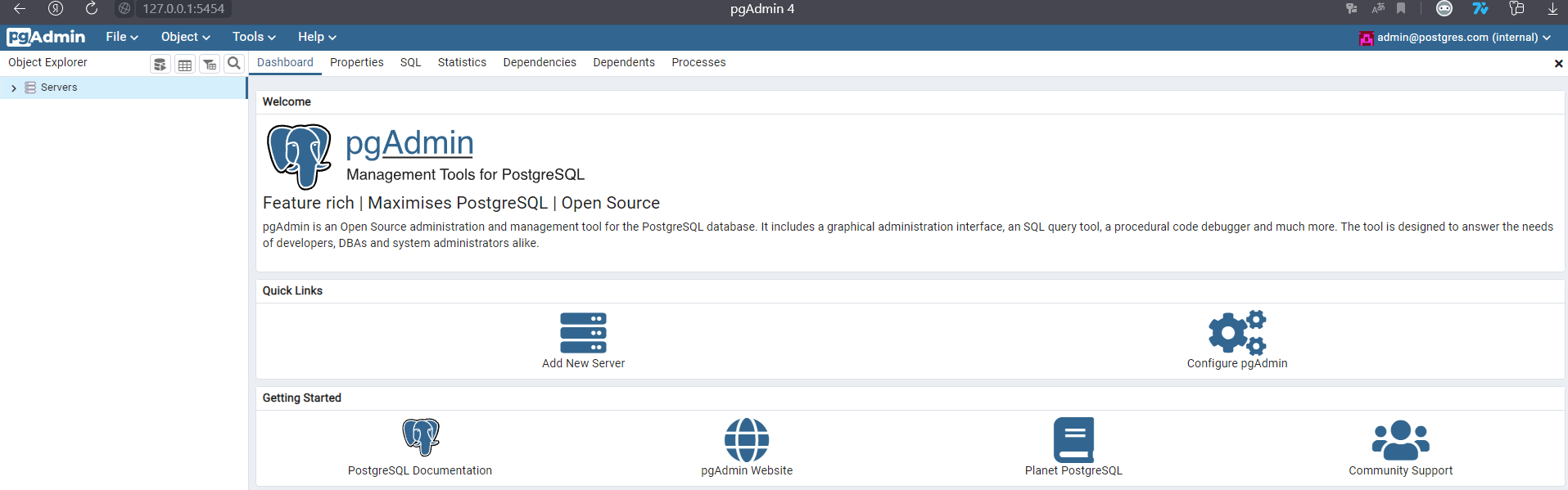
В данном случае добавилось ещё одно интересное поле: depends\_on. В него мы можем указать сервисы, которые должны быть запущены перед запуском этого сервиса. Так как pgadmin это своего рода админ панель для PostgreSQL, вполне логично, что для начала нам нужно иметь запущенную базу данных, а уже только потом запускать панель. Также в переменной порты я указал что порт 5454 связан с портом 5454 по tcp подключению внутри контейнера. Это нужно, так как pgadmin работает только с прокотолом tcp. Также, чтобы авторизоваться в эту панель, нам нужно задать пользователя, который будет администратором изначально. Для этого в наш .env файл добавляем эти строчки



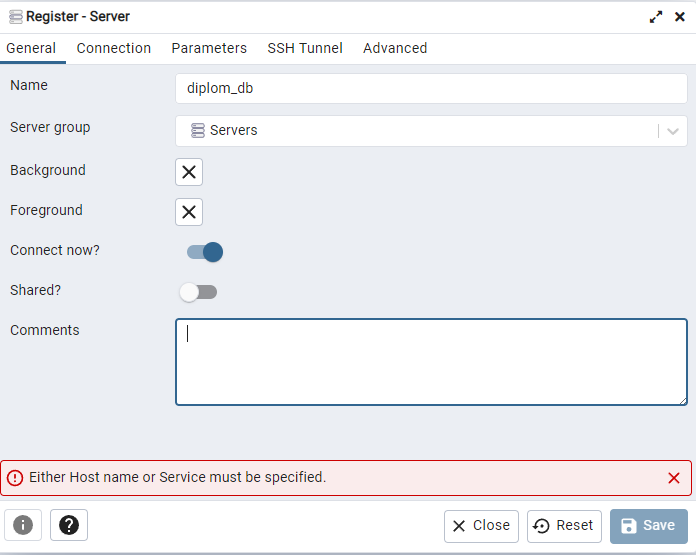
Таким образом мы сможем авторизоваться по этим данным. Поднимаем сервис командой docker-compose up --build postgres\_db pgadmin. Заходим на <http://127.0.0.1:5454> и смотрим результат:



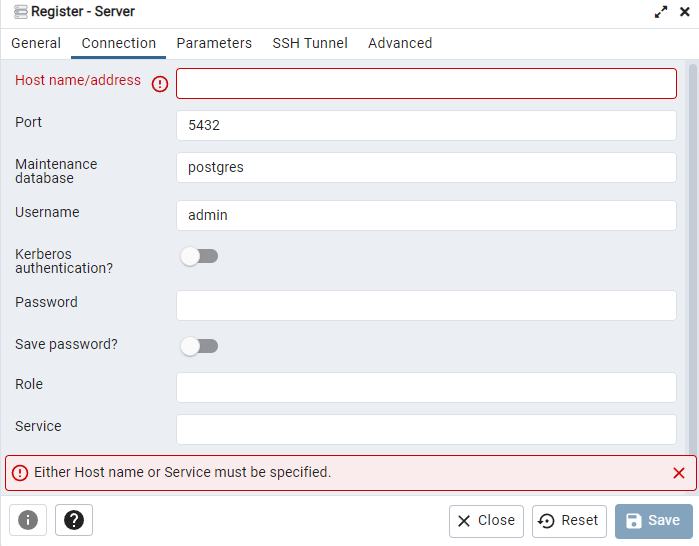
У нас открылось окно авторизации в панель. Вводим данные, которые мы указали в .env файле, проходим авторизацию, и попадаем на главную страницу pgadmin:



Для того, чтобы видеть нашу базу данных, её нужно добавить. Кликаем по Add New Server, открывается окно:



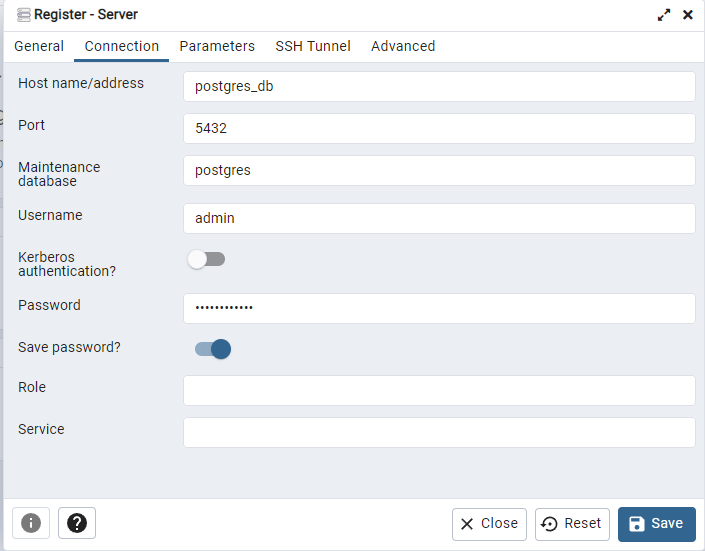
В нем мы можем указать название, коммент и настроить ещё пару параметров, переходим во вкладку Connection:



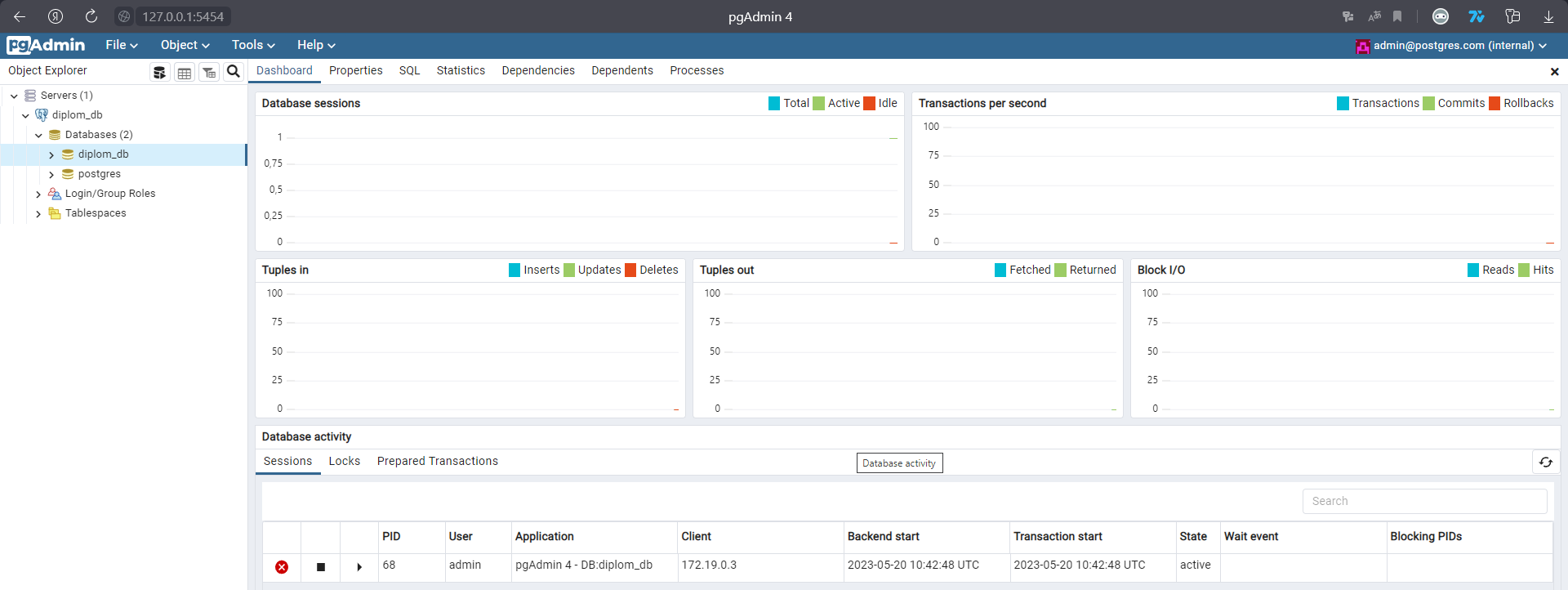
Здесь мы видим необходимые поля для подключения:

1. Host name/address – адрес нашей базы данных. Так как она запущена в контейнере, и назодится в одной связи с нашим pgadmin, мы можем указать название сервиса базы данных. В данном случае это postgres\_db. Также мы можем указать ip адрес контейнера с базой данных, чтобы его достать нужно проверять, какой сейчас ip адрес контейнера в нашей сети через команды Docker, при его смене постоянно обновлять его. Хочется нам этого? Не думаю, поэтому используем postgres\_db. До тех пор, пока pgadmin соединен в одну сеть с postgres\_db, это будет прекрасно работать
2. Mainteance database – служебная база данных PostgreSQL, по умолчанию она postgres, мы ничего не меняли на моменте создания БД, поэтому оставляем как есть.
3. Port – порт базы данных, в контейнере запущена на базовом порте 5432, поэтому оставляем 5432.
4. Username – имя пользователя в базе данных. Используем имя с .env файла.
5. Password – пароль пользователя в базе данных. Также используем с .env файла.

Остальные поля по желанию, я лишь сохранил пароль, чтобы не вводить его постоянно. Другие поля мне не понадобились. После заполения окно должно выглядеть так:



Нажимаем save и видим, что подклчюение к базе данных удалось. Мы видим наш сервер diplom\_db и его базы данных, а также статистику сервера БД.



Запуск и настройка админ панели базы данных завершена.

* 1. **Создание проекта и разработка API**

Для начала разработки нам необходимо скачать Python и IDE для редактирования кода PyCharm.

PyCharm - это интегрированная среда разработки, разработанная компанией JetBrains, специально предназначенная для разработки на языке программирования Python. Она предоставляет широкий набор инструментов и функций, которые упрощают и ускоряют процесс разработки Python-приложений.

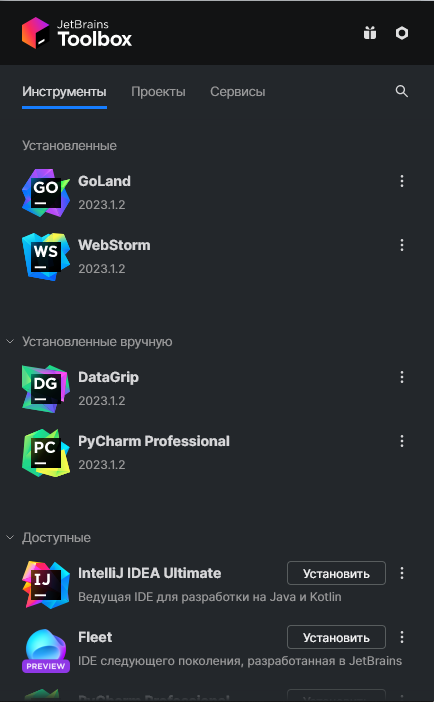
PyCharm обладает мощным редактором кода с подсветкой синтаксиса, автодополнением и интеллектуальным анализом кода, что помогает программистам писать код быстро и без ошибок. Она также обладает интегрированной системой управления версиями (git).

PyCharm предоставляет инструменты для отладки кода, выполнения модульных тестов, профилирования и анализа производительности, что помогает улучшать качество своего кода и повышать эффективность своих приложений. Она также поддерживает интеграцию с популярными фреймворками и библиотеками Python, такими как Django, Flask, NumPy, и многими другими, что делает разработку в этих окружениях более удобной и эффективной.

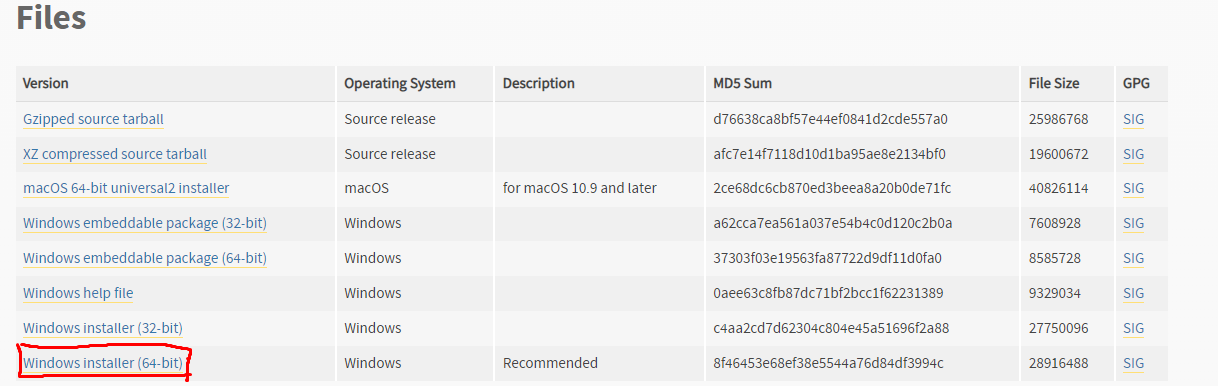
PyCharm доступен в двух версиях: Community Edition (бесплатная) и Professional Edition (платная). Professional Edition предоставляет дополнительные возможности, такие как поддержка разработки веб и мобильных приложений, инструменты для анализа данных и многое другое.

В моем случае я использую Professional Edition. Также ранее встречалась тоже лицензионная IDE DataGrip, и далее встретится IDE WebStorm. Чтобы не покупать подписку на каждую IDE, я купил подписку на все IDE от производителя JetBrains за 749 долларов в год. Для меня это приемлемая сумма, так как на моей официальной работе часто приходится использовать много IDE для работы (тестирование, DevOps, Data Science, БД, FrontEnd, BackEnd, GoLang, Java)

Для начала скачаем и установим JetBrains Toolbox, для установки всех IDE. Она доступна по адресу <https://www.jetbrains.com/toolbox-app/>. Скачиваем для Windows, устанавливаем, перезапускаем компьютер, и у нас установлен toolbox. Здесь мы видим список всех доступных продуктов. Устанавливаю нужные продукты, жду загрузки и проверяю. Теперь у меня установлены 4 IDE разработки:

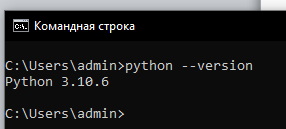


С IDE закончили, теперь нам нужен язык программирования Python для разработки. Я предпочту использовать версию 3.10.6, так как на данный момент это LTS (Latest Stable Version) из новых версий языка Python. Он доступен по ссылке <https://www.python.org/downloads/release/python-3106/>. Листаем вниз и выбираем 64 разрядный установщик для Windows:



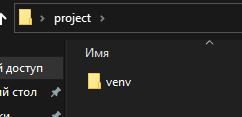
Скачиваем, запускаем, устанавливаем, перезагружаем систему. Теперь язык должен быть установлен и доступен для использования.

Чтобы это проверить мы можем запустить командную строку в любом месте и прописать python –version:



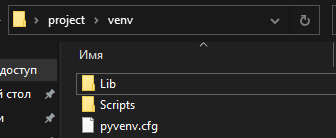
Как можно видеть, Python 3.10.6 успешно установлен. Теперь нам нужно создать наше Django приложение. Для начала я создал папку на рабочем столе, перешёл в неё, а также открыл её в терминале. Чтобы не засорять глобальные библиотеки Python, нужно создать виртуальное окружение Python – venv. Виртуальное окружение – копия интерпретатора Python, при активации виртуального окружения, текущая глобальная версия Python больше не будет никак зависеть с версией в виртуальном окружении. Все изменения окружения, такие как установка и обновление библиотек останутся только в нём. Это нужно для того, что если у нас будет несколько веб приложений, использующих разные версии библиотек, они не конфликтовали друг с другом, а имели свое рабочее окружение.

Для его создания нам нужно выполнить команду python –m venv venv (последний аргумент – как будет называться папка виртуального окружения, обычно называют venv, env или environment, кому как удобно, название не влияет на работу окружения). После выполнения у нас должна создаться папка venv.

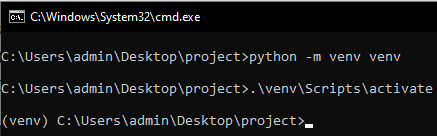


Структура этой папки состоит из 2 основных папок: Lib и Scripts. Рассмотрим их:

1. Папка Scripts: Содержит исполняемые файлы, связанные с виртуальной средой. Некоторые из этих файлов служат для активации виртуальной среды (activate.bat), для деактивации виртуальной среды (deactivate.bat) и для установки пакетов (pip.exe).
2. Папка Lib: Содержит библиотеки и модули Python, установленные внутри виртуальной среды. В этой папке находятся стандартная библиотека Python, а также любые дополнительные пакеты, установленные с помощью pip.
3. В папке venv также могут быть другие файлы, такие как pyvenv.cfg (конфигурационный файл виртуальной среды) и дополнительные файлы, связанные с установленными пакетами.

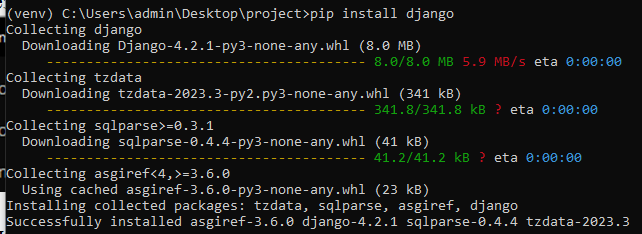


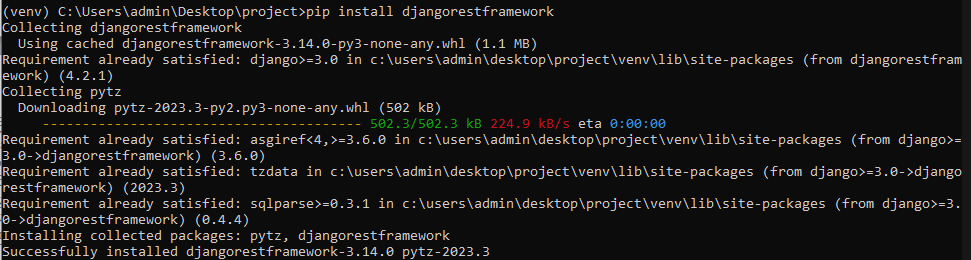
После создания виртуального окружения, нам нужно его активировать. С этим нам поможет скрипт activate.bat, находящийся по пути venv/Scripts/activate.bat. Чтобы запустить его с помощью консоли нам достаточно ввести путь к нему из корня проекта, который начинается с точки. Таким образом консоль Windows запустит скрипт по указанному пути. Команда запуска: .\venv\Scripts\activate. Выполняем и смотрим результат:



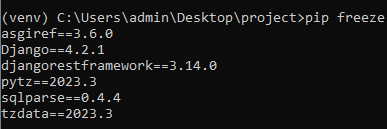
Как можно видеть, перед путем появилась надпись (venv), которая означает, что активация прошла успешна, и теперь мы запускаем команды в первую очередь от виртуального окружения. То есть, если мы запустим какую-то программу, она сначала попробует запуститься из виртуального окружения, и только потом из самого терминала. Для примера у нас есть pip.exe, менеджер установленных пакетов, если мы хотим его выполнить, например, прописав pip freeze (показывает список установленных модулей) – он выполнится из виртуального окружения и покажет модули виртуального окружения.

Теперь, когда у нас есть окружение, с которым мы можем работать, нам нужно установить Django, DRF, создать проект и настроить его. Для этого мы должны установить оба модуля командой pip install:

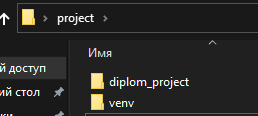




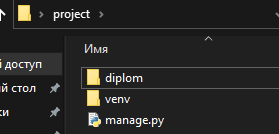
Проверяем установку командой pip freeze:



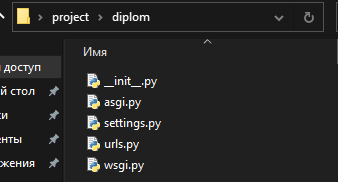
Как видим Django и DRF установлены и доступны. Остальные библиотеки – их зависимости, которые устанавливаются вместе с ними. Для того, чтобы создать наш проект нам нужно использовать команду django-admin startproject <project\_name>. Выполняем команду и смотрим результат:



У нас появилась папка diplom, в которой хранятся файлы нашего нового проекта. Так как лишняя папка нам не особо нужна, я перенёс все файлы с неё на уровень виртуального окружения. Теперь это выглядит так:



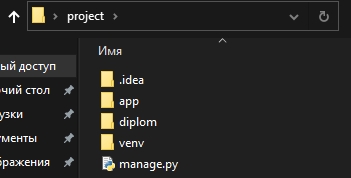
Мы видим, что у нас появилась папка с названием нашего проекта “diplom”, а также файл manage.py. manage.py служит для управления нашим приложением, его мы будем запускать вместе с нужной командой. А теперь рассмотрим папку diplom:



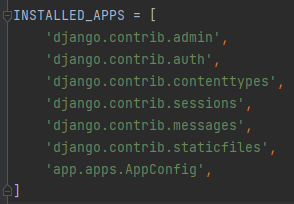
Здесь мы видим разные файлы, такие как:

1. \_\_init\_\_.py – изначально пустой файл, символизирующий что эта папка – модуль Python, который можно импортировать, написав в коде ключевое слово import.
2. asgi.py и wsgi.py – служат для развертки asgi или wsgi приложений, которые мы будем использовать в 4 главе. На данный момент они нам не интересны.
3. settings.py – хранит в себе все настройки нашего приложения. Его мы сейчас рассмотрим.
4. urls.py - хранит в себе корневой роутинг нашего приложения, его мы рассмотрим чуть-чуть позже

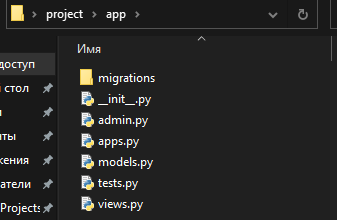
Открываем проект в PyCharm. Теперь нам нужно создать само приложение, в котором и будут описаны наши ORM модели базы данных. Для этого нам нужно выполнить команду python manage.py startapp <app\_name>. Название приложения не должно пересекаться с установленными библиотеками. Например, мы не можем создать новое приложение с названиями typing, time, django, os, sys и так далее, так как они будут пересекаться с установленными модулями. Я назвал свое приложение просто – app. Запускаем команду и смотрим результат:



Как можем видеть, наше приложение добавлено. Но оно не зарегестрированно в нашем проекте, и не видно для него. Для того, чтобы проект его видел, нужно добавить приложение в параметр INSTALLED\_APPS в файле настроект проекта settings.py, открываем. Там мы видим много настроек по умолчанию, которые нужны для работы Django проекта. Не буду рассказывать за что отвечает каждый параметр сейчас, по ходу создания мы затронем самые важные настройки. Находим переменную INSTALLED\_APPS, видим что это список установленных приложений. Есть 2 способа добавить наше приложение: добавив его название, либо путь к конфигу, который хранится в файле apps.py приложения. Я предпочитаю второй вариант, т.к. иногда конфигов может быть несколько, и в таком случае удобно определять, какой нам нужно использовать. Добавляем:

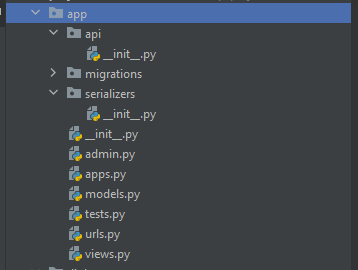


И теперь при запуске приложения Django будет видеть наше приложение, и запустит его. Теперь рассмотрим структуру нашего приложения app:



1. \_\_init\_\_.py – также говорит нам о том, что это пакет Python. В дальнейшем я больше не буду его объяснять, это правило работает всегда для любой папки, чтобы сделать её модулем.
2. admin.py – служит для настройки взаимодействия с админкой.
3. apps.py – служит для глобальных настроек приложения.
4. models.py – в нем мы будет описывать модели для базы данных, из которых Django ORM будет строить таблицы в базе данных.
5. tests.py – в нем можно создавать тесты для написанного кода или API. Желательно в больших проектах покрывать 90+% кода тестами. Но так как у нас небольшой проект, не думаю что я затрону тему тестов.
6. views.py – в нем можно создавать функции-обработчики, которые будут принимать на вход запрос, обрабатывать его и возвращать ответ.
7. migrations – содержит историю миграций. Миграции – изменения моделей в коде, с посредствующим переносом их в таблицу базы данных.

Для создания API я также добавил 2 модуля: api и serializers, а также файл urls.py. Теперь это выглядит так:



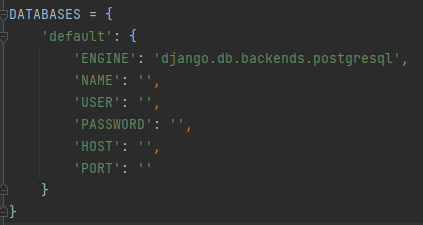
В папке api у меня будут все файлы, связанные с построением Rest приложения. В папке serializers будут различные сериализаторы для моделей из БД. Сериализация – процесс превращения одних данные в другие в обе стороны. В данном случае мы преобразовываем модель из базы данных в JSON, и наоборот. Это нужно, например, для получения какого-то объекта в базе данных и отправке понятных клиентской стороне данных этого объекта. И наоборот, например, для создания нового объекта в БД из данных, пришедших с клиентской части. Также не малую роль играет настройка нужных полей и валидация. Допустим, мы не хотим, чтобы в базе данных создавались группы без номера группы. С помощью валидации мы можем проверять входные данные, и если они подходят под наши условия – создавать объект в БД. Также может быть такое, что у нас есть поле password, и мы

точно не хотим, чтобы это поле каждый раз оправлялось на клиентскую часть. Для этого мы просто убираем это поле из сериализации. Но также может быть ситуация, когда нам нужно поле только на входе, к примеру создание нового пользователя. Для создания нам нужно имя пользователя, почта и пароль. В этом случае мы можем определить у поля атрибут write\_only, который будет только принимать значение, но не отдавать его.

Приступим к первым шагам создания приложения. Нам нужно взять модели, которые мы делали на стадии планирования приложения (в пункте 1.2). Добавляем модели в models.py. Для того, чтобы добавить их в нашу базу данных, для начала, её нужно подключить к нашему проекту. Ранее мы уже поднимали сервер с базой данных, и она доступна у нас по адресу 127.0.0.1:5432. Для того, чтобы подключиться с нашего приложения, нам нужен параметр DATABASES в нашем settings.py файле. Находим и смотрим:

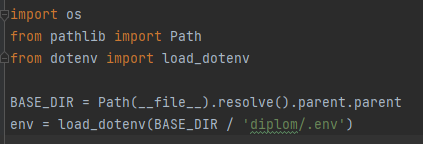


Как видим, в Django по умолчанию используется sqlite3 база данных, которая создается в корне приложения при его запуске. Нас это не устраивает, и мы хотим подключиться к нашей PostgreSQL. Для начала нам нужно изменить движок базы данных, а также добавить дополнительные поля:



В данном случае мы используем стандартный встроенный движок для postgresql, для нашего проекта его будет более чем достаточно. Теперь нам нужно заполнить название базы данных, данные для авторизации и подключения. Ранее мы уже создавали .env файл, в котором объявляли эти переменные. Поэтому будет лучше брать данные с него, чтобы при их изменении, они менялись только в одном месте.

Чтобы подключить .env файл и загрузить с него переменные, я воспользуюсь библиотекой python-dotenv. Устанавливаем командой pip install python-dotenv. После установки мы можем импортировать функцию load\_dotenv из модуля dotenv, и использовать её для загрузки переменных из нашего .env файла.

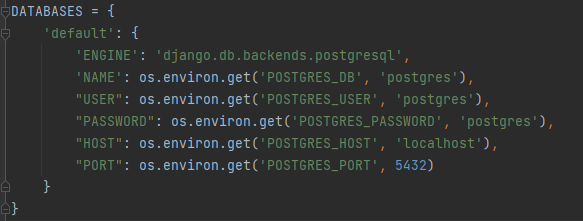


Данная функция принимает на вход путь к файлу, из которого она будет загружать переменные. Переменная BASE\_DIR генерируется при создании проекта автоматически, и хранит в себе путь до нашего проекта (в моем случае это c://users//admin//desktop//project//). Так как файл .env лежит в папке с настройками проекта, мы дописываем путь после BASE\_DIR к нему. Теперь все переменные можно получить с помощью библиотеки os:

os.environ.get(<key>, <default: Optional>)

Эта функция будет возвращать значение из нашей переменной среды, в которую мы только что загрузили значения, или, в случае их отсутствия, будет возвращать дефолтное значение, которое мы можем указать вторым аргументом.

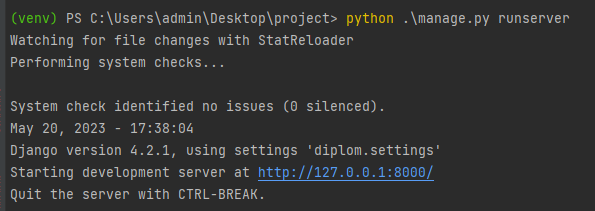
Теперь, когда мы получили доступ к данным для подключения к БД, заполняем DATABASES следующим образом:



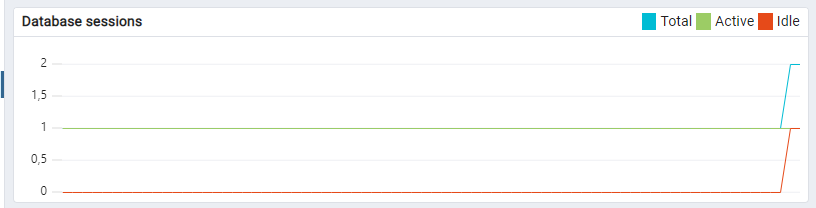
В данном случае я определил ещё базовые значения, которые при отсутствии значений в environ будут подставлены по умолчанию.

Также для подключения к БД PostgreSQL нам нужно установить дополнительный модуль psycopg2. Это библиотека, которую использует стандартный движок Djnago для подключения. Существует 2 версии этой библиотеки: psycopg2 и psycopg2-binary. Разница заключается в том, что psycopg2 на самом деле пакет Python, осуществляющий привязку к библиотеке C libpq, которая используется для взаимодействия с базой PostgreSQL. Что означает, что для её использования нам может потребоваться устанавливать нужные зависимости вручную. binary версия содержит все зависимости в коробке, и запустится без дополнительных требований. Это нам точно подходит, поэтому устанавливаем командой pip install psycopg2-binary

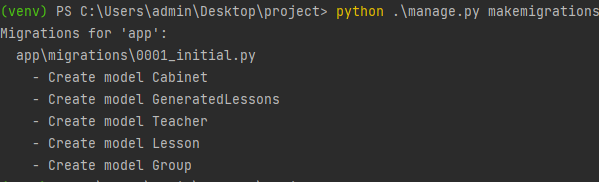
Теперь при запуске приложения, если мы посмотрим в pgadmin, который поднимали ранее, у нас должно прибавиться подключение к базе данных со стороны Django приложения. Проверим. Для запуска самого простого веб-сервера мы можем использовать команду runserver. Вводим python manage.py runserver (мы запускаем manage.py вместе с агрументом runserver. manage.py выполнит нашу команду, если такая есть в нашем django приложении)



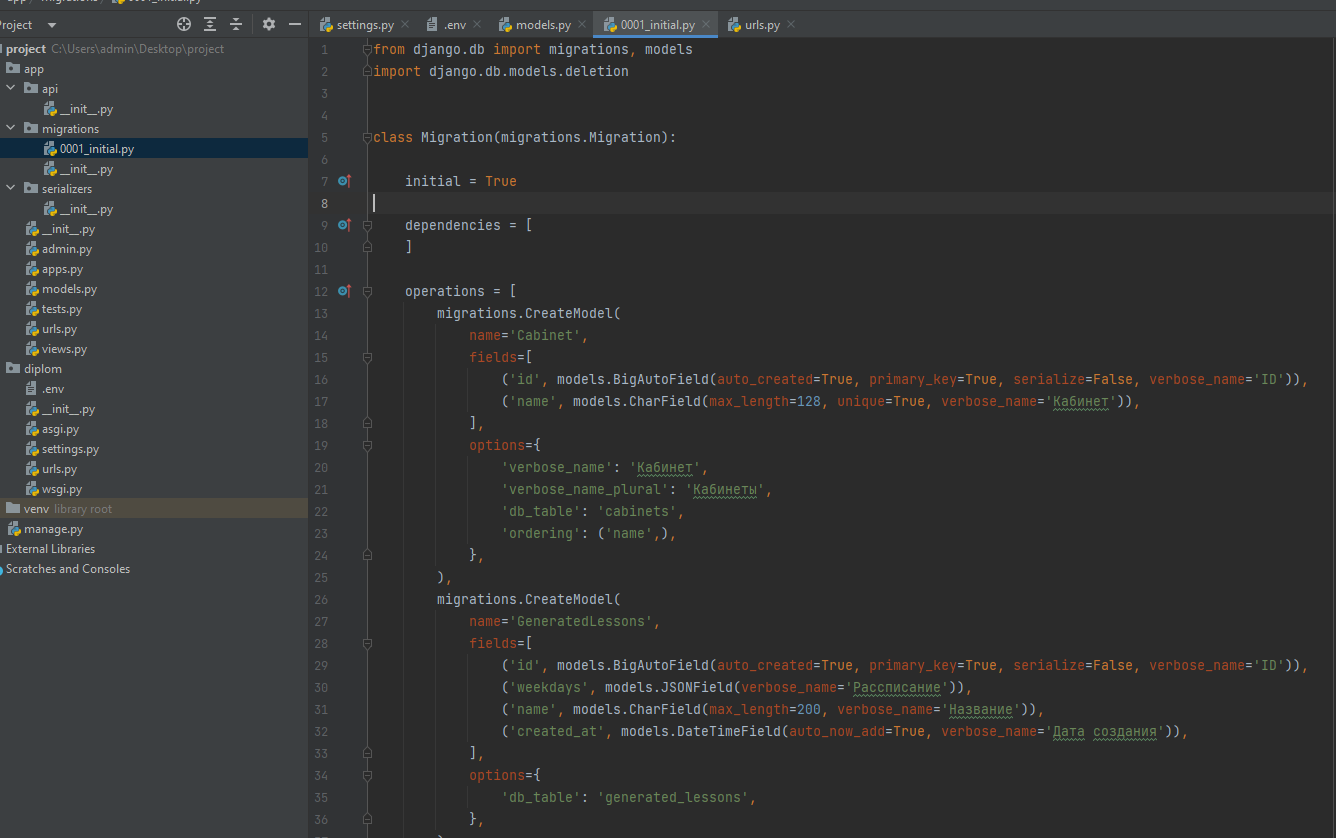
Видим что сервер успешно запустился, заходим в pgadmin, и видим по графе подключений, что у нас появилось новое подключение:



Но в данный момент, если посмотреть на таблице в базе данных, их не будет, соответственно наши ORM модели не будут работать. Для того, чтобы они заработали и перенеслись в базу данных, нам нужно выполнить первую миграцию. Миграция создается командой makemigrations. Запускаем python manage.py makemigrations и смотрим результат:



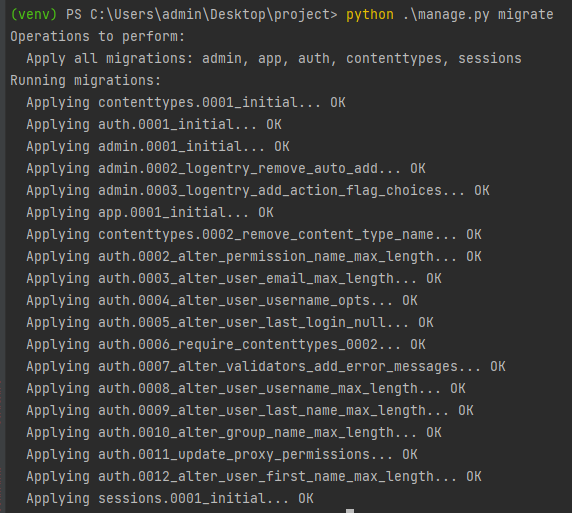
Как можем видеть, Django атоматически увидел изменения в моделях и создал для них миграции в базу данных. Теперь при каждом изменении полей в модели ORM мы должны делать и применять миграцию в БД, чтобы изменения перенеслись и туда. Взглянем на файл миграций:



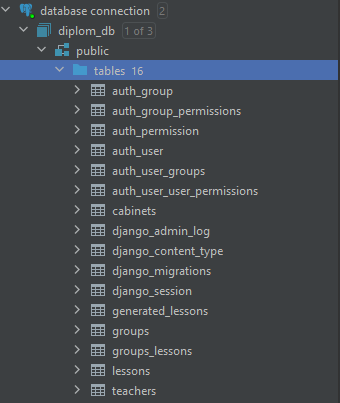
Здесь мы видим, что миграции – это классы, унаследованные от класса django.db.migrations.Migration. Они генерируются автоматически при изменении модели и запуска команды makemigrations, но также мы можем создать свою собственную миграцию. Всё что нам нужно сделать – создать новый .py файл с названием XXXX\_NAME.py, где XXXX – порядковый номер миграции, а NAME – название миграции. Далее унаследовать класс миграции, добавить 2 переменных

dependencies и operations. В dependencies мы обязательно должны указать миграцию, после которой будет запускаться эта миграция. В operations мы передаем список операций, которые будут выполнены в рамках транзакции в базу данных.

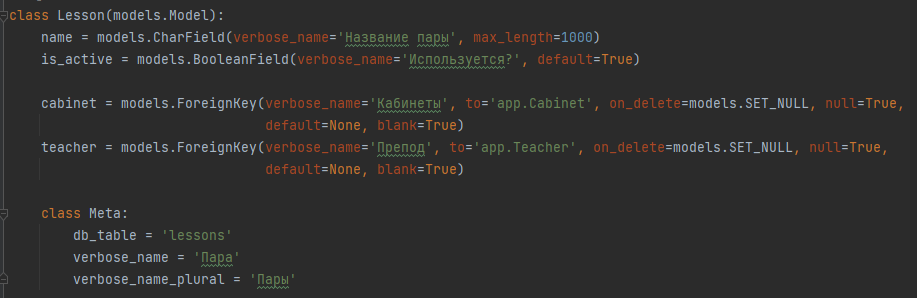
С миграциями разобрались, теперь нужно применить их. Запускаем команду python manage.py migrate, ждем результат:

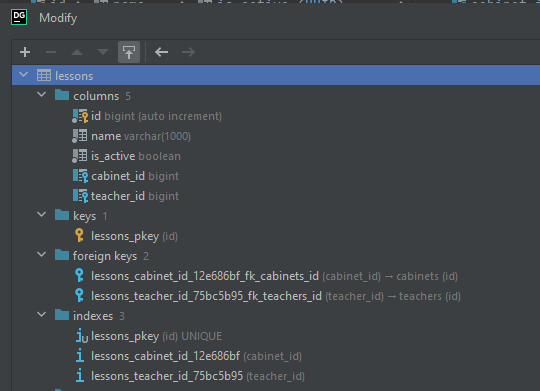


Видим что помимо нашей миграции появилось ещё куча других миграций. Дело в том, что помимо нашего приложения, у Django по умолчанию есть свои приложения, которые установлены по умолчанию (админка, авторизация, сессии и т.д.). У них тоже есть свои модели, и при их разработке разработчики Django также делали свои модели и миграции. Они и выполняются вместе с нашими в новую базу данных, так как там нет таблиц, нужных для их работы. Теперь мы можем посмотреть базу данных, и увидеть, что все таблицы создались:



Для примера посмотрим на таблицу lessons и нашу модель lessons:



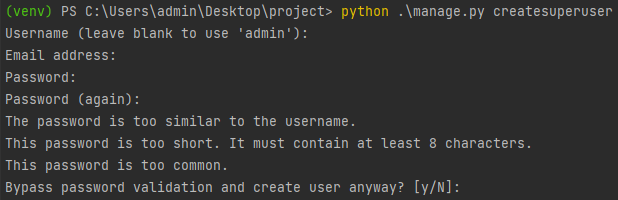


Как мы видим, миграция автоматически создала таблицу, которая соответствует нашей ORM модели. Также, миграция по умолчанию добавляет столбец id, который является первичным ключом для любого объекта в БД. Ключом может быть любое уникальное поле. Ещё у нас есть 2 ключа к связи с другой таблицей. Это сделано для того, что если объект по этому ключу был удален, у нас запустился определенный тригер, который сделает что-то с этим значением. К примеру: если мы удалим какой-то пост в социальной сети, для освобождения памяти мы можем каскадно удалить все лайки и просмотры, которые хранятся по ключам в других таблицах. Или допустим у нас есть платежный провайдер, он работал с клиентами год, и перестал работать. При его удалении мы бы не хотели стирать историю пополнения пользователей, а поставить пустое значение, или например значение провайдера по умолчанию. Для этого в ORM мы определяем поле on\_delete, которое и будет отвечать за тригер при удалении связанного объекта. В конце мы видим индексы, которые автоматически создались. Индексы – технология базы данных, которая позволяет ускорить выполнение запросов по этим полям в БД. По умолчанию индексы – бинарное дерево, в котором поиск нужного элемента будет происходить в сотни раз быстрее, если объектов в БД много. Хоть в нашей БД и не будет много объектов, не упомянуть это я не мог. К примеру, у меня на основной работе таблица с просмотрами пользователей хранит в себе 1трл записей и весит больше 1 терабайта. Поиск в такой таблице без индексов занял бы около дня.

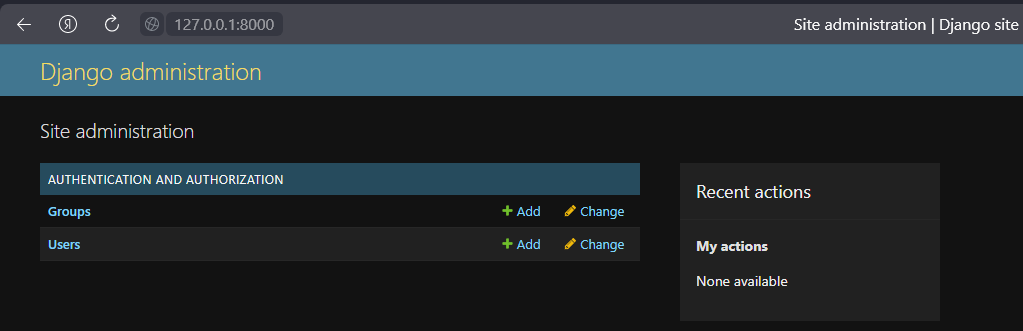
Теперь, когда у нас есть модели в базе данных, мы можем настроить админку, чтобы управлять ими. Как мы помним, это настраивается в файле admin.py нашего приложения. Для начала зайдем в админку. Для того чтобы зайти туда, нам нужен пользователь с соответствующими правами. Так как мы ещё не создавали пользователя, самое время это сделать. Вводим команду python manage.py createsuperuser и смотрим результат:



Нас просят ввести имя пользователя, или оставить пустым. По умолчанию будет admin (название нашей учетной записи пользователя windows). Я оставлю пустым. Далее просят ввести почту, также оставлю пустой, так как она нам не пригодится. Затем ввожу пароль admin и повторяю его.

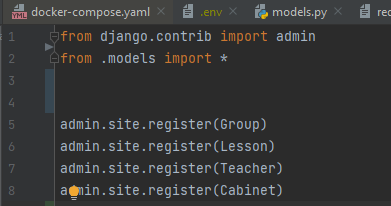


Пароль при вводе скрывается, поэтому его не видно. Команда нам пишет, что пароль слишком похож на имя пользователя, он слишком короткий и слишком простой. Так как мы работаем локально, нам это не особо важно. Но если мы будем развёртывать приложение для всех – очень важно сделать надежные данные для авторизации. Вводим “y”, и видим что пользователь создан. Запускаем сервер командой runserver, заходим на <http://127.0.0.1:8000/admin/> и видим окно авторизации. Вводим данные созданного пользователя и заходим в главное меню админки.

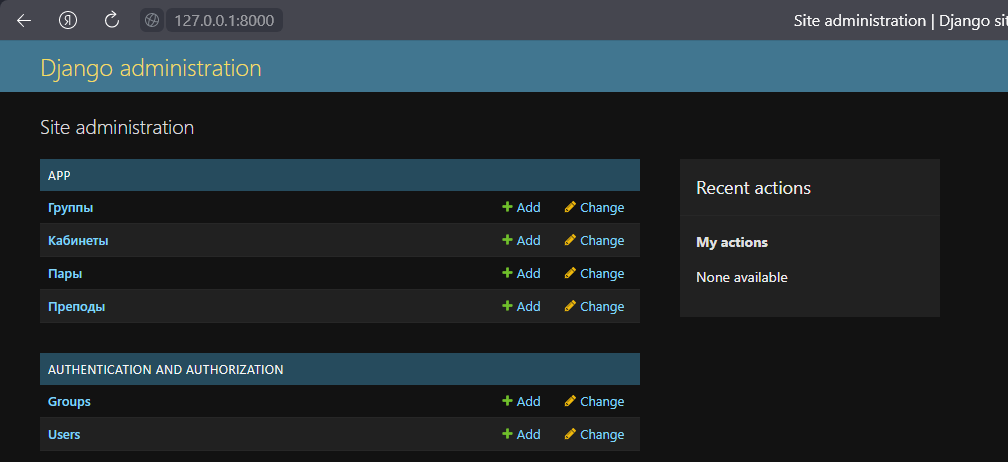


Как мы видим, наших моделей в этом списке нет. Чтобы их добавить, самое просто что мы можем сделать – зарегистрировать их в admin.site.redister(<model>)

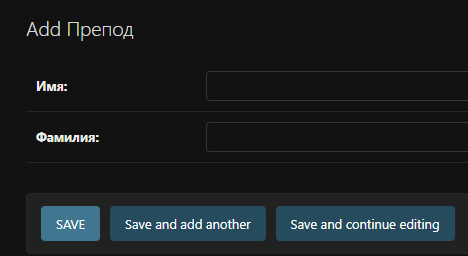
Для этого мы импортируем все модели из класса models и регистрируем их.



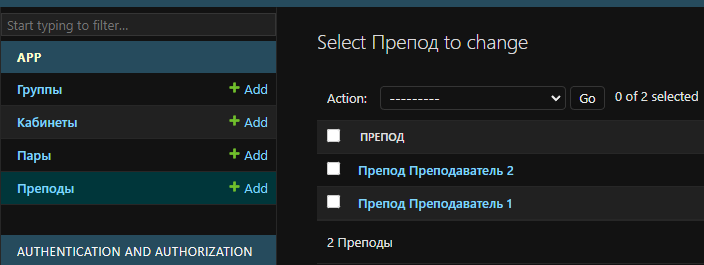
Перезагружаем приложение и смотрим.



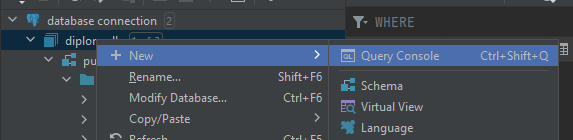
Как видим, наши модели появились в админке, и мы можем ими управлять. Для примера я создам по 2 кабинета и преподавателя, 3 предмета и 2 группы. Приступаем. Для начала нажимаем кнопку добавить на преподавателях, попадаем на форму добавления:



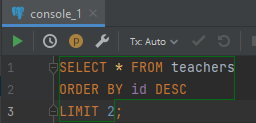
Мы видим 2 обязательных поля, как мы и указывали в модели преподавателя. Также мы видим 3 кнопки: сохранить, сохранить и добавить ещё один объект, сохранить и продолжить редактирования. Первая сохранит данные и перекинет на список моделей из базы данных, вторая сохранит и перекинет на новую страницу создания объекта. Третья сохранит и оставит на текущей странице. Я добавлю 2 преподавателя «Преподаватель 1» и «Преподаватель 2». После сохранения, при переходе в список преподавателей в админке, мы видим что 2 объекта успешно были созданы:



Давайте проверим нашу базу данных на наличие этих записей. Для этого мы вновь подключимся к ней с помощью DataGrip, нажмем на нашу базу данных diplom\_db и выберем действие «New > Query console»



Нам откроется новая консоль, в которой мы можем писать запросы. Так как мы сохраняли преподавателей, нам нужно достать записи с таблицы teachers. Пишем SQL запрос в базу данных для получения этих данных:



Разберем эту запись:

SELECT – ключевое слово, которое говорит нам о том, что мы хотим выбрать объекты с базы данных, и получить их поля.

\* - означает что мы хотим получить все поля, которые есть у этого объекта. Если мы хотим получить определенные поля, например только id и имя, мы можем заменить \* на id, first\_name.

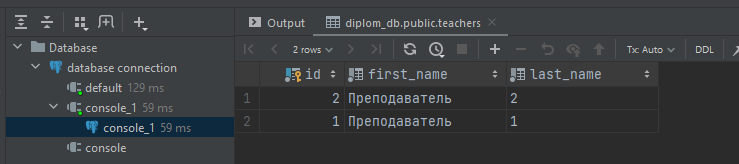
FROM teachers – означает что мы используем таблицу teachers

ORDER BY – сортировка полученных результатов по условию

id DESC – условие сортировки, которое означает, что мы хотим получить объекты в порядке убывания их id. Зачастую это означает, что мы получаем объекты по новизне. Каждый раз, когда мы сохраняем новый объект в эту базу данных, значение id присваивается этому объекту и увеличивается на 1, чтобы присвоить его следующему объекту.

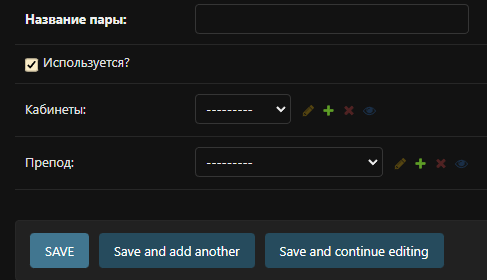
LIMIT 2 – означает что мы хотим получить только первые 2 записи. Так как мы создали только 2 записи, это не обязательно, но если бы у нас уже существовали другие записи, мы бы увидели не только 2, которые сейчас создали.

Запускаем выполнение запроса и смотрим результат:



Видим, что наш запрос обработан за 59 миллисекунд, и в нашей базе данных действительно создались 2 объекта, которые мы сохраняли в админке. Таким же образом создаем пару кабинетов.

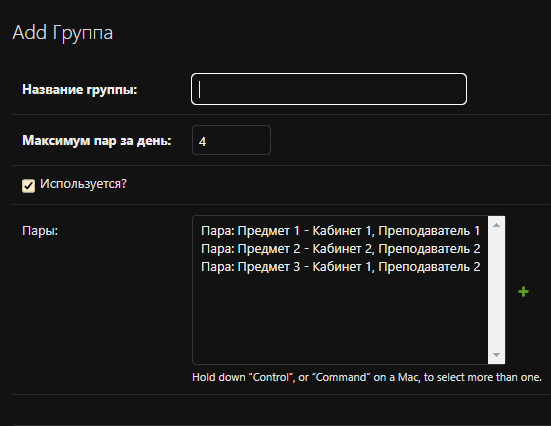
Теперь создадим предметы. Посмотрим на форму создания:



Здесь у нас есть название, будет ли предмет использоваться для генерации расписания. А также 2 поля Кабинет и Преподаватель. Это связь с моделями, которые мы создавали ранее. Например Преподаватель 1 будет вести в 101 кабинете, или Преподаватель 2 будет вести в 302 кабинете и т.д. Для примера я создам 3 предмета:

1. Предмет 1, 1 кабинет, преподаватель 1
2. Предмет 2, 2 кабинет, преподаватель 2
3. Предмет 3, 1 кабинет, преподаватель 2

Теперь перейдем к созданию групп. Смотрим форму создания:



Как видим, нас здесь просят указать название, сколько максимум пар в день может иметь группа, а также привязать созданные предметы, которые есть у группы, чтобы при генерации можно было выбрать нужное кол-во, которое будет расставлено на неделю. Для первой группы я выберу все 3 предмета, а для второй только 2 и 3 предмет. Также у второй группы будет максимум 2 предмета в день.

После создания моделей нам нужно начать проектировать REST API приложение. Для начала разберемся, что такое REST API.

REST API - это архитектурный стиль и набор принципов, которые используются для создания веб-сервисов, обеспечивающих коммуникацию и обмен данными между клиентскими приложениями и серверами. В REST API приложении ресурсы представляют собой конкретные объекты или данные, к которым можно получить доступ через определенные URL-адреса (на языке программирования они называются эндпоинтами). Клиентские приложения могут выполнять различные операции над этими ресурсами, такие как создание, чтение, обновление и удаление (CRUD – Create, Read, Update, Delete).

Основные принципы REST включают:

Использование стандартных методов HTTP: REST использует GET, POST, PUT и DELETE, для выполнения операций над ресурсами. Например, GET используется для получения данных, POST - для создания новых ресурсов, PUT - для обновления существующих ресурсов, а DELETE - для удаления ресурсов.

Использование уникальных эндпоинтов: Каждый ресурс в REST API имеет свой уникальный эндпоинт. Например, /users может быть эндпоинтом для получения списка пользователей, а /users/1 - для получения конкретного пользователя по ключу 1.

Без состояния: Клиент и сервер взаимодействуют без сохранения состояния между запросами. Каждый запрос должен содержать все необходимые данные для выполнения операции, и сервер не должен сохранять информацию о предыдущих запросах.

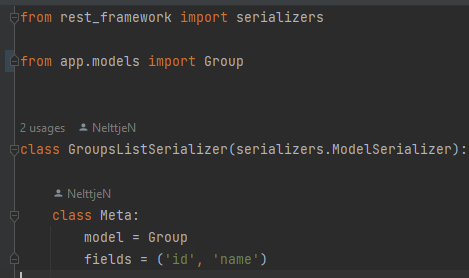
Использование формата данных: REST может использовать различные форматы данных для представления ресурсов, такие как JSON или XML. JSON является наиболее популярным форматом для передачи данных в REST API. Его мы и будем использовать.

REST API позволяет создавать гибкие и масштабируемые веб-сервисы, которые могут быть использованы различными клиентскими приложениями. Он обеспечивает стандартизацию и упрощает разработку и интеграцию приложений, позволяя им взаимодействовать и обмениваться данными эффективным способом.

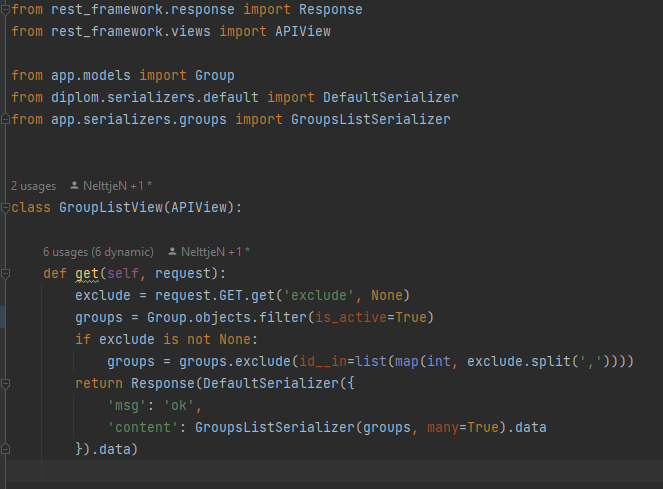
Изначально Django не предоставляет эффективных инструментов для создания REST приложения. Поэтому ранее мы устанавливали библиотеку Django Rest Framework, которая является надстройкой над Django, и позволяет создавать простой и удобный REST API.

Для создания первого эндпоинта, который будет отдавать список групп для генератора расписания, мы должны создать новый файл в папке api. Назовем его groups.py.

Чтобы создать ресурс – нам нужно унаследовать класс от APIView пакета rest\_framework.views. В нем мы создадим функцию get, которая будет означать метод GET эндпоинта, к которому будет привязан ресурс. Так как ресурс будет отдавать информацию о моделях Group, нам нужна сериализация этих моделей. Создадим файл groups.py в папке serializers. Для создания сериалайзера мы используем наследование от класса ModelSerializer пакета rest\_framework.serializers. Также определим у него подкласс Meta, в котором укажем какую модель и поля мы хотим получать на выходе. Получаем такой результат:



Теперь напишем сам ресурс. В нем добавим параметр exclude, который может приходить в GET запросе как Query Params, которые хранятся в запросе в GET. Получаем такой ресурс:

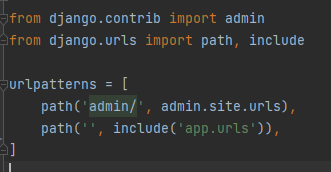


Здесь мы отправляем в ‘content’ список групп, которые мы вытащили с базы данных.

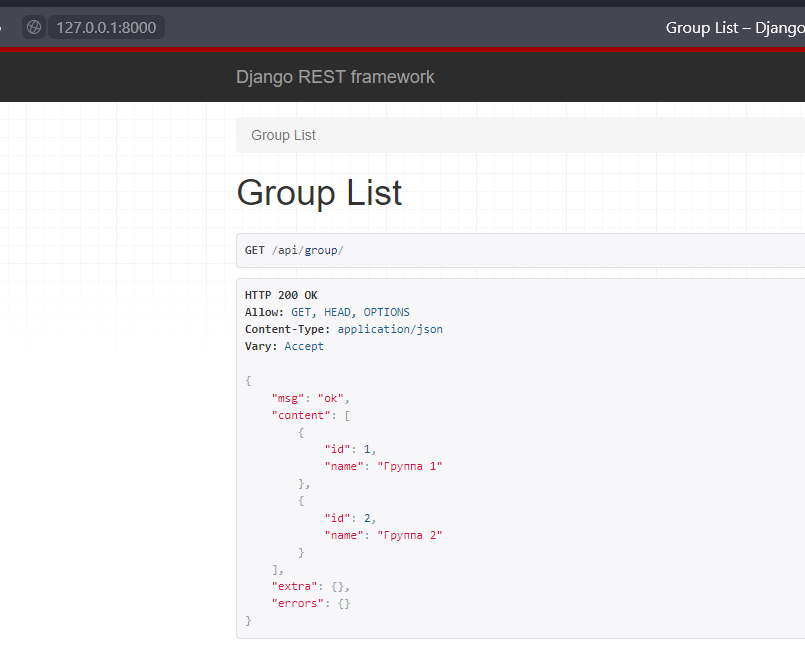
Чтобы добавить эндпоинт и привязать к нему этот ресурс, добавим в файл urls.py списки apipatterns и urlpatterns. В apipatterns мы будем добавлять все эндпоинты, в urlpatterns мы будем добавлять обычные ссылки, которые будут отрисовываться обычными функциями с файла views.py. Получаем такой результат:



Теперь нам нужно зарегистрировать этот файл в нашем корневом файле urls.py, в настройках приложения. Открываем его, и добавляем в существующий urlpatterns строку:



Мы используем include, который автоматически подставит все эндпоинты с файла app.urls с переменной urlpatterns, которую мы только что создали. Теперь при запуске наш ресурс должен быть доступен по адресу <http://127.0.0.1:8000/api/group/>. Заходим и проверяем ответ:



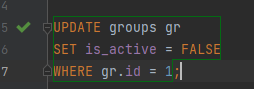
Как видим нам отдается удобный список групп в понятном для нас формате. Допустим мы добавили первую группу в генератор, и нам нужно получить список оставшихся групп. Добавляем параметр exclude и смотрим результат <http://127.0.0.1:8000/api/group/?exclude=1>



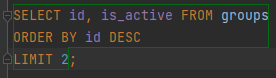
Как можем заметить, группа с id 1 убралась из списка. Попробуем теперь убрать и вторую группу, отправив запрос <http://127.0.0.1:8000/api/group/?exclude=1,2>



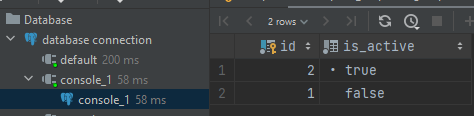
Как видим, всё прекрасно работает, группы убираются. Но также мы добавляли параметр “используется” в модели группы. Уберем его у первой группы. Для этого нам нужно поменять поле is\_active у группы с id 1 на False. Чтобы это сделать, напишем SQL запрос в базу данных:



Выполняем, пишем запрос на получение этого поля у групп:



Выполняем его и смотрим результат:

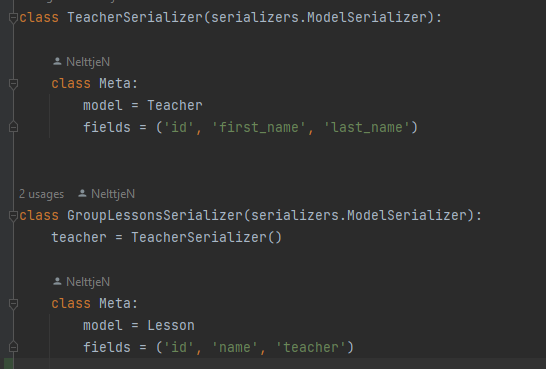


Как видим, группа теперь неактивна. Попробуем отправить запрос на <http://127.0.0.1:8000/api/group/>. В этот раз он нам должен выдать только вторую группу, так как первая группа скрыта. Заходим и проверяем результат:

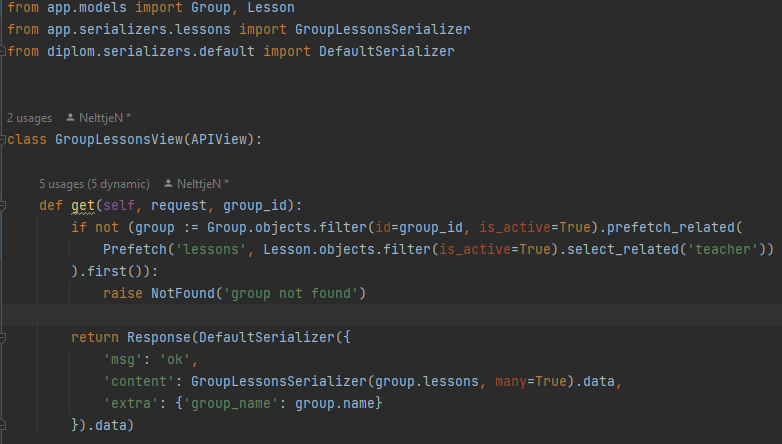


Всё работает как надо, группа успешно скрыта из списка групп, доступных по намешу эндпоинту.

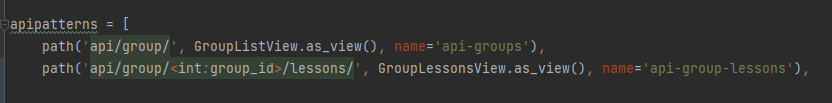
С группами разобрались, теперь предположим, что пользователь выбрал группу, и мы должны показать ему список предметов группы, для его настройки. Для этого нам нужен новый эндпоинт. Создаем новый файл lessons.py в api и serializers папках. Для начала напишем сериализатор. Нам нужны из урока его название, и имя преподавателя. Так как преподаватель у нас это связь с моделью Teacher, для модели Teacher тоже нужно будет сделать сериализацию. Иначе у нас будет отправляться айдишник, а не информация о преподавателе. Пишем 2 сериализатора:



Здесь мы переопределяем поле teacher и указываем, что оно у нас использует сериализатор, а не обычное IntegerField поле. Теперь пишем нужный ресурс.



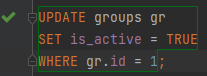
Здесь мы добавляем в get функцию новый аргумент: group\_id. Так как мы будем получать список уроков с помощью айди группы, оно будет отправлено в эндпоинте, а из него будет передано в ресурс. Пробуем найти группу по такому айдишнику, если группа не найдена – отдаем 404 ошибку, и пишем, что группа не найдена. Также мы используем функцию prefetch\_related и класс Prefetch. Это делается для оптимизации запросов в базу данных. Каждый объект lessons будет вытащен из базы данных по нужным критериям. Нам не нужно будет проходить по каждому уроку, доставать ему преподавателя (select\_related у предмета подтянет преподавателя вместе с ним, используя SQL синтаксис LEFT OUTER JOIN). Далее мы просто отправляем в сериалайзер предметы, и указываем что мы хотим получить список предметов (many=True), а не только один предмет. Добавляем в urls.py наш новый эндпоинт с указанием <int:group\_id>. Этим мы укажем маршрутизатору, что мы ожидаем число, и суем его как аргумент group\_id в функцию.



Проверяем результат. Помним, что мы скрыли группу с id 1. Для начала проверим, что группа, которая неактивна, не находится:



Как и ожидалось, получаем ответ HTTP 404 и ошибку что группа не найдена. Теперь вернем группе что она активна SQL запросом в базу данных

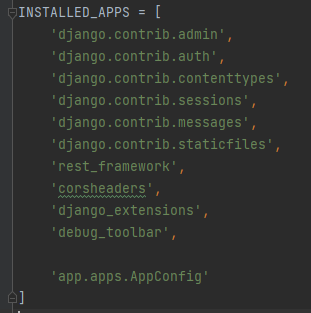


Выполняем, перезагружаем страницу, смотрим результат:



Как видим, мы получили всю необходимую информацию и ответ HTTP 200.

Также проверим запросы в базу данных, чтобы удостовериться, что select\_related и prefetch\_related работают как нужно. Для этого мы установим новый модуль в наш проект. Устанавливаем командой pip install django-debug-toolbar. Чтобы наша дебаг панель работала, нам нужно добавить её как приложение в INSTALLED\_APPS в settings.py. Переходим и добавляем.



Также ранее я забыл упомянуть, что нам нужно добавить rest\_framework и corsheaders в INSTALLED\_APPS, для полноценной работы нашего REST API. С rest\_framework понятно, но что за corsheaders? В REST приложении есть такое понятие как CORS (Cross-Origin Resource Sharing) - это механизм, который контролирует доступ к ресурсам на веб-странице из другого источника (домена), отличного от источника, который предоставил исходную страницу. По умолчанию браузеры применяют политику безопасности, называемую Same-Origin Policy, которая запрещает JavaScript-коду загружать ресурсы с другого домена. Это ограничение не позволяет клиентскому приложению, работающему на одном домене, делать запросы к API, размещенному на другом домене, если не разрешено. Если говорить проще – у нас есть счет в банке, на котором 10000р, а у банка есть, к примеру эндпоинт /api/send/<user\_id>/?amount=10000. Предположим, мы зашли на подозрительную страничку, на которой есть кнопка «Получить 1 миллион рублей». Мы на неё нажимаем, и он нашего браузера JavaScript код, вшитый в страницу, отправляем запрос на этот эндпоинт, указав user\_id администратора. Так как мы проходили авторизацию в банке, при отключенном CORS этот запрос обработается, и наши деньги уйдут, хотя мы были даже не на сайте банка. Для этого и нужно проверять, с какого ресурса пришёл запрос. Если он пришёл с какого-то другого сайта – мы его блокируем, и не позволяем деньгам нашего клиента утечь в кошелек злоумышленника. В общем CORS блокирует запросы злоумышленников с других источников, тем самым обеспечивая безопасность нашему REST API приложению.

Для настройки CORS используется 3 переменных, добавляемых в settings.py файл:

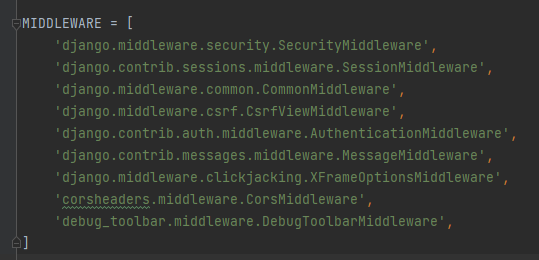
1. CORS\_ORIGIN\_ALLOW\_ALL – bool значение, которое при true просто отключает эту технологию, принимаю любые запросы с любых источников. В реальных приложениях это ВСЕГДА должно быть отключено, и отключено по умолчанию. Но в нашем проекте, чтобы оно не мешало, я поставлю True и отключу его.
2. CORS\_ORIGIN\_WHITELIST – Список разрешенных доменов для подключения. Например [“http:localhost:8000”, “http://127.0.0.24:3000”] и т.д.
3. CORS\_ALLOW\_METHODS – Список методов, которые разрешены на нашем сервере. Если мы, к примеру, укажем список из [“GET”, “POST”, “DELETE”], но не добавим “PUT”, то любые PUT запросы будут блокироваться CORS из любых источников, даже с тех, которые находятся в белом списке.

Как я сказал в первом пункте, я отключу CORS используя эту строчку в нашем settings.py файле



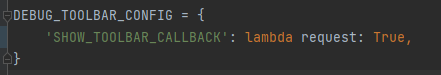
В реальных проектах так делать КАТЕГОРИЧЕСКИ ЗАПРЕЩЕНО! Нужно обязательно указывать белый список доменов, с которых у нас будут обрабатываться запросы.

Вернемся к Debug панели. Чтобы она работала, нам нужно также добавить её обработчик в MIDDLEWARES переменную. Также мы должны добавить и CorsMiddleware, чтобы у нас корректно работал CORS. Добавляем:

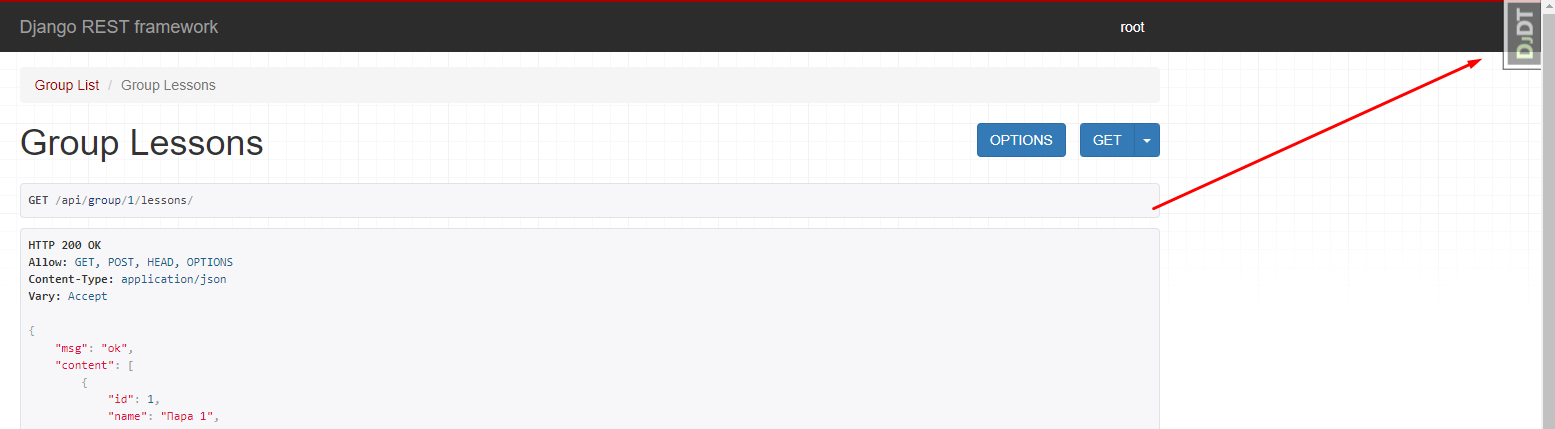


Теперь о том, что такое middleware. Простыми словами Middleware – прослойка между нашим сервером и клиентом. Запрос, приходящий от пользователя, сначала проходит все Middleware. Там может быть авторизация, валидация каких-то данных, проверка куки, и всего того, что мы должны сделать перед работой с самим запросом, и формированием ответа. Также работает и в обратном порядке, наш ответ от сервера сначала проходит все Middleware, и только потом улетает обратно к клиенту.

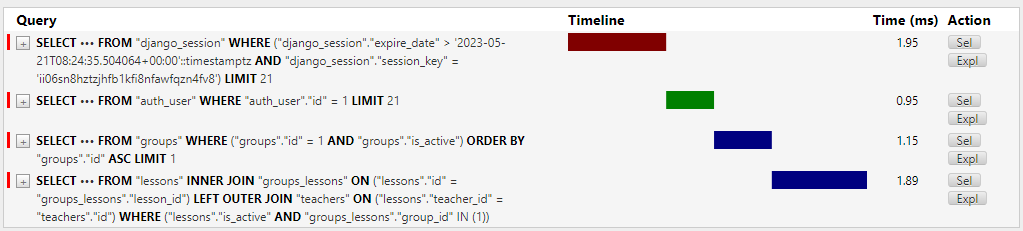
С этим разобрались, но если мы обновим страницу, всё ещё не увидим никакой Debug панели. Для этого нам нужно определить ещё одну переменную в settings.py, DEBUG\_TOOLBAR\_CONFIG. Это словарь из различных настроек нашей дебаг панели. Один из таких параметров - SHOW\_TOOLBAR\_CALLBACK. В качестве значения, мы должны положить функцию, которая будет принимать запрос. И возвращать True или False, что будет означать показывать панель, или нет. По умолчанию, всегда возвращается False, поэтому мы и не видим панель. В идеале мы должны проверять права пользователя, откуда подключение. Но в качестве ознакомительных целей мы просто будем показывать её всегда. Для этого напишем следующее:



Как видим, мы передали lambda функцию, которая принимает запрос, и всегда возвращает True. Теперь при перезагрузке страницы мы обнаружим панель в правом верхнем углу. Проверяем:



Панель появилась, открываем её и видим множество вкладок, которые могут помочь нам в отладке приложения. Я не буду рассказывать что делает каждая вкладка, для наших целей нам хватит одной вкладки SQL, в которой пишутся все запросы в базу данных, которые были выполнены в рамках этого запроса. Открываем и видим это:



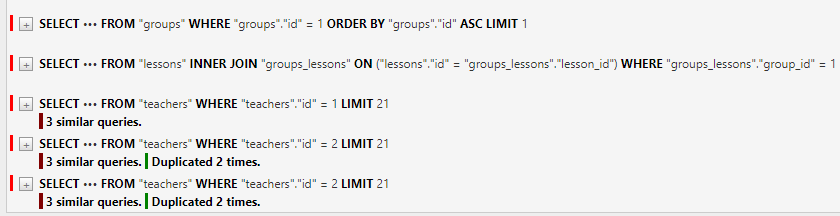
Здесь мы видим все запросы, время выполнения, а также можем посмотреть их поподробнее, или использовать SQL запрос EXPLAIN, который покажет план выполнения выбранного запроса.

Первые 2 запроса выполняет авторизация Django и Django Rest Framework. Это как раз выполняется в тех самых middleware, и авторизовывает пользователя перед обработкой запроса.

Далее мы видим 2 наших запроса, один на группы, второй на предметы. С группами всё понятно, просто достаем по id. Куда интереснее посмотреть предметы:



Здесь мы видим, как работает тот самый prefetch\_related. Он делает отдельный запрос в таблицу lessons, присоединяет с помощью INNER JOIN значения из таблицы group\_lessons (где хранится связь группа - предмет), где id предмета равен id предмета в таблице group\_lessons, затем с помощью select\_related с помощью LEFT OUTER JOIN у нас присоединяется значения преподавателя к каждому уроку, и наши уроки фильтруются что они используются и принадлежат выбранной группе. И всё это делается в одном запросе. Если мы уберем полностью prefetch\_related, и перезагрузим страницу, увидим это:



Как можем видеть, для каждого предмета у нас отдельно идет запрос на преподавателя, а также нет фильтрации по активному предмету. Если в рамках нашего проекта это всего 3 запроса, то представим нагруженный проект. Например у нас есть пост, у которого 10000 лайков. И к каждому лайку нам нужно вытащить имя пользователя, поставившего лайк. В таком случае у нас будет 10002 запросов в базу данных, вместо 3. А теперь ещё представим, что у нас одновременно посты смотрят 100000 пользователей. Сможет ли наша база данных отработать такие запросы? Теоретически да, если мы потратим весь бюджет компании на базу данных. Но разве оно нам нужно? Не думаю. Поэтому оптимизация запросов очень важна в высоконагруженных проектах.

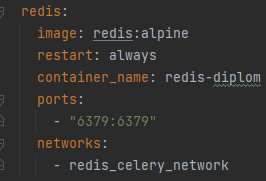
С эндпоинтами мы закончили, теперь у нас всё готово для разработки алгоритма генерации рассписания: редактирование моделей в админке, получение всех групп и предметов группы. Переходим к разработке алгоритма.

* 1. **Разработка алгоритма генерации расписания**

Для работы нам потребуется в первую очередь добавить ещё 2 сервиса – redis и celery. Redis представляет из себя NoSQL базу данных, которая часто используется в качестве брокера сообщений или кеша. Мы будем её использовать и для кеша, и как брокер для celery.

Celery предоставляет простой способ создавать задачи, которые выполняются асинхронно в фоновом режиме, а не в основном потоке выполнения приложения. Это особенно полезно для задач, которые требуют значительного времени выполнения или которые не должны блокировать основной поток приложения, такие как отправка email письма, вычисление какого-то алгоритма, и т.д.

Для начала установим оба этих модуля. Выполним команды pip install redis, pip install celery. Теперь поднимем redis сервер, для этого возвращаемся в наш docker-compose файл и добавляем новый сервис.



Я использовал официальный image redis’а, доступный на Docker Hub: <https://hub.docker.com/_/redis>. Со всеми ключеввыми словами сервиса мы уже знакомы, поэтому переходим к добавлению redis к нашему Django приложению.

Так как мы будем использовать его ещё и как кеш, добавим новую переменную CACHES в нашем settings.py файле.

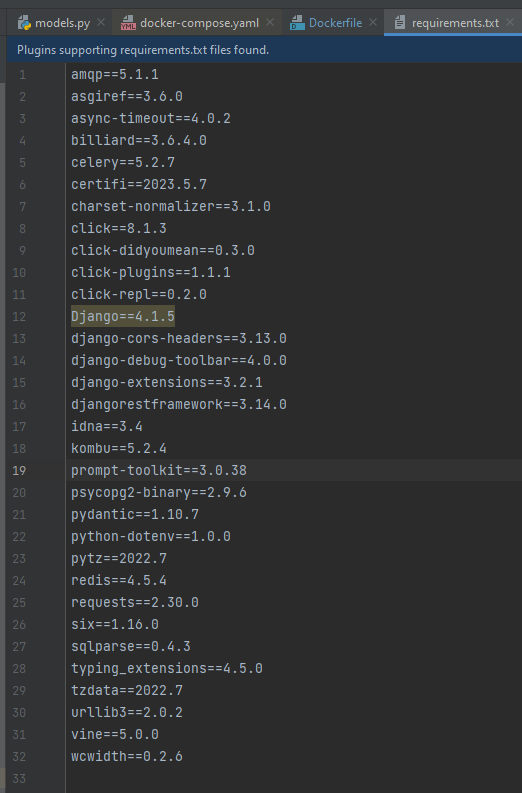


Для начала я определил хост и порт redis сервера. После этого я добавил кеш redis, который будет использоваться по умолчанию. Так как мы работаем на версии Django 4.2, у него есть встроенный Backend для redis кеша. Если бы мы работали с версией Django <4.0, нам пришлось бы устанавливать дополнительную библиотеку redis-cache для этого. Далее LOCATION – адрес, по которому бекенд будет поключаться к redis серверу. Важно обратить внимание, что здесь мы используем не протокол http, а протокол redis. Также в OPTIONS мы можем указать различные параметры, о которых можно почитать в документации. Я лишь указал, что кеш будет использовать 2ую базу данных, а 0 и 1 я оставлю для Celery. Теперь при перезапуске приложения у нас есть кеширование. Кешей, как и баз данных может быть несколько. Например redis кэш и memcached кэш. Они отличаются подходом к кешированию, и другими асектами. Чтобы добавить новый кеш, нам нужно добавить новый ключ, по которому мы сможем получить этот кеш, и также прописать бекенд и остальные настройки для него. Все кеши хранятся в django.core.cache, мы можем импротировать cache и caches переменные. Caches содержит в себе все кеши, которые подключены к нашему Django приложению, а cache содержит в себе кеш, который указан под ключом default. В нашем случае это будет redis.

Теперь приступим к созданию Celery сервиса. Так как Celery запускается от нашего Django приложения, нам нужно собрать ещё и его. Для этого в корне приложения создаем Dockerfile. В нем нам нужно описать процесс сборки нашего приложения. Для запуска Django приложения нам нужно в первую очередь все модули python, которые мы установили, а также сам python.

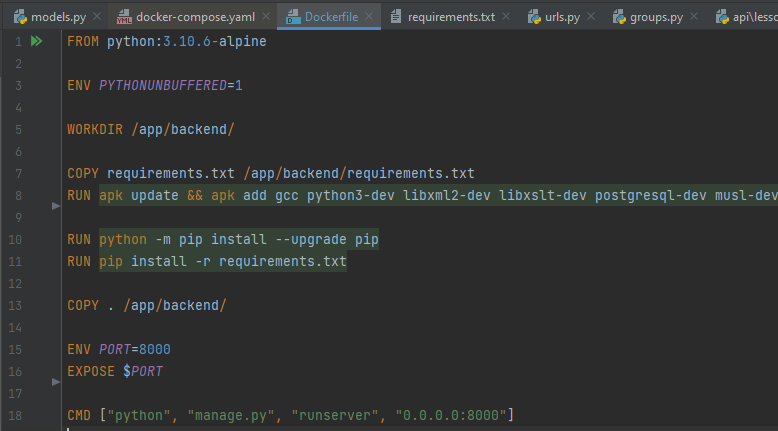
Для начала выгрузим все модули, которые мы установили. Для этого мы можем использовать команду pip freeze. Но эта команда просто напишет в консоль список всех зависимостей нашего проекта. Чтобы сразу записать их в файл, а не копировать

и вставлять в него, мы должны добавить в конце знак >, который озаначает запись результата выполнения команды в файл, и название файла, в который будет записано. В Python принято называть файл с зависимостями requirements.txt, мы так и поступим, выполняем команду pip freeze > requirements.txt и видим, что у нас создался файл. Открываем его и видим список всех зависимостей с их версиями.



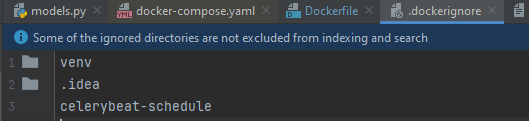
Версии пишутся для того, чтобы если вышли обновления на наши модули, они не повлияли на работу проекта. Может выйти новая версия, в которой поменяли все методы, и она не будет совместима с версией, на которой писался проект. Поэтому важно зафиксировать версии на этапе их добавления, и обновлять вручную, если необходимо.

Теперь начнем описывать Dockerfile. Как я и писал ранее, нам нужен Python. Берем его также из официального image c Docker Hub <https://hub.docker.com/_/python> Так как я работаю на версии 3.10.6, она нам и нужна в контейнере. Поэтому я выбрал версию 3.10.6-alpine. Теперь напишем Dockerfile:



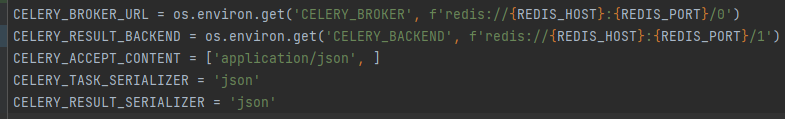
1. FROM python:3.10.6-alpine – указываем, что этот Dockerfile выпоняется после Dockerfile в image python:3.10.6-alpine
2. ENV PYTHONUNBUFFERED=1 - По умолчанию, Python буферизует вывод стандартных потоков, что означает, что вывод может быть отложен и не отображаться сразу же. Установка этого значения рекомендуется, чтобы гарантировать более точное и мгновенное отображение вывода, улучшить анализ логов и обеспечить лучший отклик приложения в среде контейнера.
3. WORKDIR /app/backend/ - указываем, что мы находимся в папке /app/backend/ в консоли контейнера.
4. COPY requirements.txt /app/backend/requirements.txt – копируем файл из нашей локальной папки в папку в контейнере. Первый аргумент – местолопожение файла относительно того, где лежит Dockerfile, второй аргумент – путь в контейнере, куда мы будем копировать этот файл и под каким названием.
5. RUN apk update && apk add gcc python3-dev libxml2-dev libxslt-dev postgresql-dev musl-dev – команда, которую мы запускаем на этапе сборки контейнера. В данном случае – обновить все пакеты Linux и установить список других пакетов, которые понадобятся для запуска Django проекта.
6. RUN python -m pip install --upgrade pip устанавливаем обновления для менеджера пакетов Python – pip
7. RUN pip install -r requirements.txt – устанавливаем все модули, которые мы выгружали в requirements.txt файл.
8. COPY . /app/backend/ - копируем все файлы из папки, в которой находится Dockerfile (т.к. первый путь указан “.”, а это указывает на текущую папку) в папку /app/backend/ внутри контейнера
9. ENV PORT=8000 и EXPOSE $PORT – открываем порт 8000 для других контейнеров
10. CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"] – команда, которая будет выполнятся по умолчанию при запуске контейнера. Если в docker-compose мы укажем ключевое слово command у сервиса, который использует этот Dockerfile – эта строка будет игнорироваться.

Также, при копировании приложения, я бы не хотел, чтобы он копировал всю папку с виртуальным окружением, так как она нужна нам только локально, содержит в себе довольно много файлов, и может занять долгое время для копирования. Для этого создадим .dockeringore файл вместе с Dockerfile. В него мы поместим папки и файлы, которые будут игнорироваться Docker’ом.



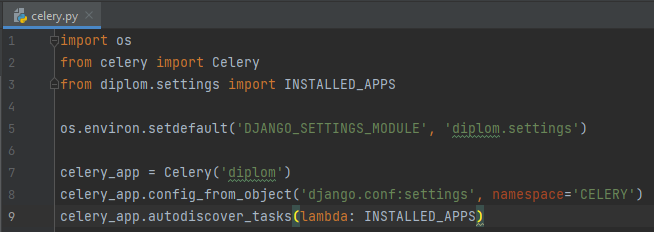
Я добавил виртуальное окружение, папку для работы IDE и файл с задачами Celery.

Теперь, когда у нас есть Dockerfile для нашего Django приложения, пришло время развертывать Celery сервис. Перед этим добавим Celery в наш Django проект. Открываем settings.py и добавляем строчки для работы Celery:



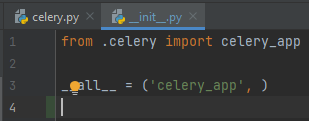
Указываем что брокер у нас это 0 база данных redis’a по умолчанию, либо брать из env файла. С результатами аналогично, но будем использовать 1ую базу данных, чтобы не мешать очереди сообщений. Также указываем что мы работаем с json объектами как на входе, так и на выходе результатов.

Теперь создаем файл celery.py в папке с настройками нашего Django приложения, и вписываем в него следующие строчки:

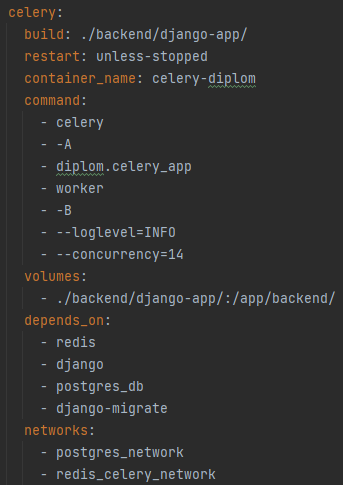


Здесь мы создали celery\_app, привязаный к нашему Django проекту diplom, загрузили настройки из файла settings.py. Используем namespace, чтобы указать, какие переменные нам нужно читать. Он загрузит все переменные из settings.py файла, которые называются CELERY\_... Ну и ставим обработчик задач на все наши установленные в Django приложения.

Теперь заходим в \_\_init\_\_.py файл настроек нашего проекта, и импортируем туда celery\_app, чтобы запуск Celery видел наш celery\_app, и подтянул с него задачи и настройки.



Мы установили Celery в наш Django проект. Приступим к запуску его в Docker. Для этого добавим новый сервис в docker-compose файл.

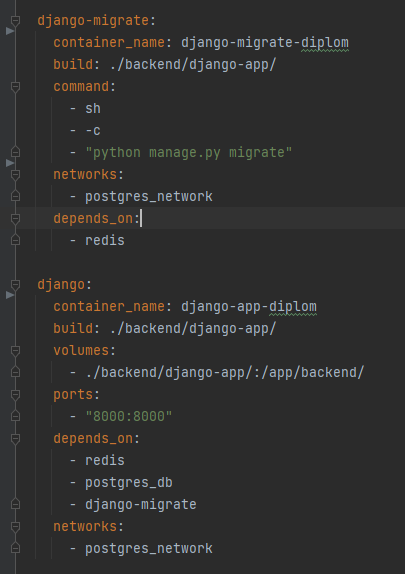


Вместо image мы пишем build, так как мы создали свой Dockerfile, а не берем готовый c Docker Hub. В build мы указываем локальный путь к Dockerfile относительно docker-compose файла.

В зависимости ставим чтобы сначала запустились Django, PostgreSQL, Redis и прошли миграции в базу данных. Без этих сервисов запуск может получиться раньше, и сервис упадет, так как будет пытаться подключиться к ещё не запущенным контейнерам. Также мы подключаем Celery сразу к двум сетям: postgres\_network и redis\_celery\_network, ведь нам нужно как подключение к БД, так и подключение к redis для работы.

С помощью command мы изменяем команду с запуска проекта Django на запуск Celery. Ещё благодаря volumes мы указываем, что все изменения файлов в каталоге нашего Django проекта будут изменяться и внутри контейнера.

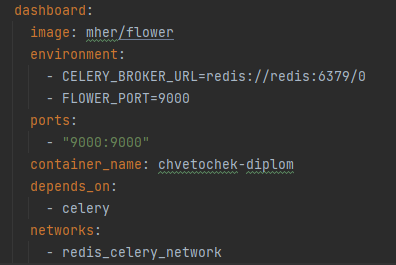
Ну и добавим ещё 2 сервиса: наш Django проект и миграции к нему, чтобы не забывать их выполнять.



Также я ещё подключу модуль flower, позволяющий мониторить состояние тасков и воркеров Celery. Беру его с Docker Hub

<https://registry.hub.docker.com/r/mher/flower>

И добавляю в docker-compose



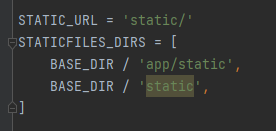
Вместо .env файла параметры окружения я прописал сразу, с помощью environment. Не забываем подключить его к нашей сети redis\_celery\_network.

После всего этого, наш docker-compose файл для всего бекенда готов. При его запуске у нас запустятся сразу все сервисы: PotgreSQL, PgAdmin, Redis, Django, Celery, Flower. Но эта версия только для разработки. Для развертки проекта позже мы будем использовать gunicorn и wsgi приложение, вместо обычного runserver.

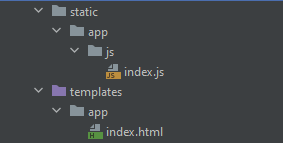
Теперь проработаем страницу настройки алгоритма. Для этого я буду использовать обычный JavaScript. Для самого начала нам нужно хранить html страницу и javascript файлы к ней. Для этого заходим в settings.py. Для того, чтобы Django видел шаблоны, нужно указать путь к ним. Находим переменную TEMPLATES и добавляем туда пути к нашим шаблонам.



Также делаем и со STATICFILES

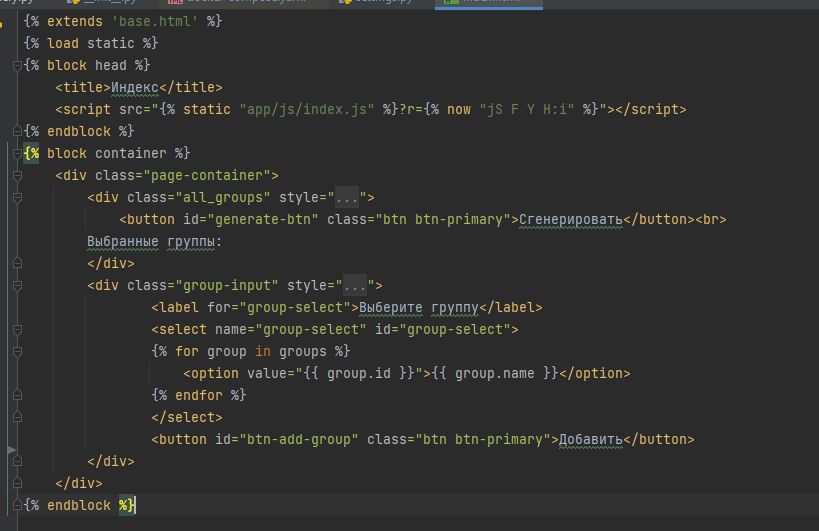


После этого я создам index.htm и index.js файлы в templates и static папках.



Принято в этих папках также создавать папку с названием приложения, в котором они находятся. Всё потому, что если будут файлы с одинаковым названием, они перезапишутся при сборке static файлов и шаблонов. Но если мы разделяем их папками, всё будет работать как надо.

В index.html я добавлю небольшую форму, в которой будет выбор списка групп, кнопка выбрать, и финальная кнопка сгенерировать рассписания. В рендер я также буду передавать список начальных групп с базы данных. Сам html будет унаследован от base.html, который был описан в пункте 1.2. Весь html код выглядит так:



С помощью {% static <path> %} мы получаем ссылку на наш index.js файл, с помощью которой браузер получит его. ?r={% now %} я использую, чтобы избежать кэширования этого файла браузером, и все изменения происходили в реальном времени.