

Geekbrains

**Разработка CRM системы для сервисной компании на языке
Python и фреймворках Django и Django Rest Framework**

IT-специалист:
Программист Python Цифровые профессии
Лобанова А.С.

Москва
2024

СОДЕРЖАНИЕ

Оглавление	
ВВЕДЕНИЕ	4
Назначение и цели создания CRM системы	7
Структура приложения и требования к функциональности	7
Формирование базы данных приложения	10
Установка и настройка Django Rest Framework	11
Что такое Django Rest Framework	11
Установка и настройка	14
Настройка административной панели	17
Кастомизация панели администрирования	18
Описание структуры и функций приложений проекта	19
1. Приложение «user»	19
Модели	19
Сериализаторы	20
Представления данных модели	24
2. Приложение «staff»	27
Модели	27
Сериализаторы	29
Представления данных модели	35
3. Приложение «client-service»	39
Модели	39
Сериализаторы	40
Представления	44
4. Приложение «Common»	49
Миксины для моделей	49
Миксины для сериализаторов	51
5. URLs	52
6. Пагинация	53
7. Итоговое представление REST API	55

Дальнейшее развитие проекта	57
ЗАКЛЮЧЕНИЕ.....	58
СПИСОК ЛИТЕРАТУРЫ.....	59

ВВЕДЕНИЕ

Что такое CRM — определение и расшифровка

Customer Relationship Management (CRM) — система, которая помогает выстроить отношения с клиентами и следить за совершаемыми сделками.

Программа собирает информацию о клиенте в электронную карточку, учитывая все действия:

момент заявки → консультацию со специалистом → визит в магазин → покупку → доставку

Работая в CRM, менеджер наглядно видит историю взаимодействия с каждым клиентом. В ней отображаются все сообщения и разговоры, ФИО, номера телефонов, email. Однако CRM выступает не только как хранилище данных. Она также подсказывает менеджеру или владельцу бизнеса, как лучше связываться с тем или иным клиентом, какова скорость ответа и качество сделки. Система собирает всю аналитику в отчеты, помогает работать с воронкой продаж, планировать дальнейшее взаимодействие с покупателями.

Используя CRM, вы больше не будете вручную составлять клиентские базы в электронных таблицах. Программа все сделает за вас, а также напомнит, когда клиенту пора отправлять оповещение о заказе или информацию о новинках.

CRM — большой комплекс функций, включающий маркетинг, техподдержку, продажи, аналитику, коммуникацию с клиентами. Система автоматизирует основные процессы бизнеса.

Данная система будет строиться под нужды небольших сервисных компаний, занимающихся продажами оборудования, запчастей и поддержкой гарантийных обязательств.

Выбор CRM это ответственный шаг, определяясь с которым стоит учитывать не только текущие цели, но и то, что нам может потребоваться в перспективе. В этом плане фреймворк Django выступает универсальным

инструментом для разработки веб-приложения. Вне зависимости от того, что вам необходимо в настоящее время, в будущем уже существующий ресурс, можно изменять, дополнять, видоизменять.

Определимся с понятием, что такое фреймворк? Академия Яндекса говорит нам, что это готовый набор инструментов, который помогает разработчику быстро создать продукт: сайт, приложение, интернет-магазин, CMS-систему. Звучит неплохо, а что там с Django. Django считается один из самых популярных.

Благодаря такой гибкости фреймворк завоевал сердца многих компаний. В настоящее время на данной платформе функционируют такие гиганты, как: YouTube, Google, Pinterest, Reddit и другие.

Почему компании выбирают Django в качестве фреймворка для написания больших и сложных проектов. Отметим основные преимущества Django:

1. Всё включено – Django со старта предлагает разработчику большое разнообразие уже установленных инструментов и библиотек.
2. Развитая система – структура Django позволяет нам собирать ресурс, как конструктор;
3. Долгая история сервиса – Django на рынке разработки функционирует уже более 18 лет, за которые сервис успел развиваться в высокоуровневую платформу;
4. Расширяемость – большое количество библиотек и плагинов предоставляет даёт возможность быстро добавлять новые программные модули;
5. Библиотеки – с помощью библиотек мы получаем готовые решения реализации сложных задач. К популярным библиотека Django можно отнести: Django REST Framework для работы с API, Pillow – для работы с изображениями, Django-rest-swagger – для автоматической генерации документации.
6. Административная панель – ресурс автоматически генерируется при создании приложений. Отсутствие необходимости создавать вручную

административную панель – весомый аргумент при выборе инструмента разработки;

7. SEO-дружественность – в Django мы легко можем реализовать требующиеся функции для настройки поисковой оптимизации.

8. ORM - в Django представлено объектно-реляционное отображение, благодаря которому происходит взаимодействие приложение с базами данных.

Django написан на языке программирования Python, что полностью отражается в его структуре. В основе лежит архитектура MVT – Model-ViewTemplate – модель-представление-шаблон.

В данной архитектуре в моделях мы описываем объекты, их свойства и функции. За счет моделей происходит создание и добавление объектов, их обновление и удаление.

В представлениях решаются задачи обработки HTTP-запросов, исполнения бизнес-логики с помощью прописанных методов и свойств, формирование ответов на запросы.

Если модели и представления – это бэкенд, то за фронтенд отвечают шаблоны. В Django продумана собственная система подключения HTML-шаблонов. Их главная функция в отличие от представлений не исполнение логики, а отображение данных. Стоит отметить, что шаблоны могут быть, как статическими, так и динамическими.

Возможность создавать программный продукт, написав всего несколько строк кода, очаровывает. Это обуславливает популярность Django в среде разработки и наш выбор в качестве дипломного проекта темы «Разработка CRM системы для сервисной компании на языке Python и фреймворках Django и Django Rest Framework».

В своей дипломной работе мы ставим перед собой следующие задачи:

1. Изучение фреймворков Django и Django Rest Framework и инструментов его работы
2. Разработка системы CRM
3. Выработка умений и навыков проектирования структуры базы данных

Назначение и цели создания CRM системы

За 5 лет своей трудовой деятельности я сменила несколько различных компаний. Все они занимались похожей деятельностью. Мне показалось интересным создать приложение, которое будет управлять процессами продаж оборудования и запчастей, а также взаимодействием с клиентами по вопросам ремонта или пуско-наладки оборудования. В данной дипломной работе я создала только лишь небольшой макет системы CRM. В дальнейшем планирую его развивать и добавлять функционал.

Структура приложения и требования к функциональности

При запуске проекта на Django на старте автоматически создается директория с вложенными файлами. Структура проекта при первом запуске имеет следующий вид:

- myproject/
 - manage.py
 - myproject/
 - __init__.py
 - settings.py
 - urls.py
 - asgi.py
 - wsgi.py

С помощью файла manage.py мы будем осуществлять управление проектом – запускать сервер, создавать новые приложения, выполнять миграции

информации в базу данных, создавать администратора, создавать и менять пароль и много другое.

Далее идет пакет Python одноименный с нашим проектом, в примере выше это пакет с названием «myproject». В моем же проекте он носит название «config». Как пакет его определяет наличие файла `__init__.py`. Данный пакет содержит стандартные файлы, которые используются для настройки проекта:

- `settings.py` – файл, в котором прописываются настройки проекта. Включает в себя описание путей к приложениям, настройки статики, важные параметры, связанные с работой подключаемых модулей и настроек безопасности.

- `urls.py` – в данном файле прописываются маршруты приложения. По адресам, прописанным в данном файле, будет строиться навигация на нашем ресурсе. Несмотря на то, что данный файл может содержать все ассоциации url адресов с представлениями, принято разделять его на части и в каждом приложении создавать свой файл `urls.py`.
- `asgi.py` – модуль, описывающий связь проекта с веб-сервером через интерфейс ASGI.
- `wsgi.py` - модуль, описывающий связь проекта с веб-сервером посредством интерфейса WSGI.

Если стартовый пакет содержит файлы, отвечающие за общие настройки проекта, то за функциональная логика прописывается в приложениях.

Приложение в Django представляет собой отдельный фрагмент функциональности сайта. Оно может, как реализовать работу всего сайта, так и отдельной функциональности (раздела или подсистемы). При разработке структуры сайта рекомендуется выделять приложения по выполняемому спектру задач

В нашем проекте мы выделяем следующие приложения:

- Приложение «api» - данный пакет будет отвечать за представление всех запросов Django Rest Framework. Поможет в этом библиотека drf-spectacular. Также здесь мы создадим директорию spectacular, в которой будут храниться настройки этой библиотеки.
- Приложение «config» — это основное приложение проекта, в нем хранятся основные настройки.
- Приложение «common» - это приложение будет содержать в себе различный функционал, общий для всего проекта. Его можно применить в любом из приложений.
- Приложение «users» - этот пакет будет отвечать за создание новых пользователей, авторизацию и аутентификацию в системе.
- Приложение «staff» - пакет, отвечающий за создание сотрудников компании. А именно будут создаваться 2 типа сотрудников: менеджеры и сервис-работники. Он связан с таблицей базы данных «users».
- Приложение «clients» - пакет, в котором будет прописана логика работы с базой данных клиентов: добавление клиентов, сохранения информации о клиентах в базе данных.
- Приложение «equipment» - приложение, отвечающее за создание товаров в базе данных, таких как оборудование, зап.части и тд.
- Приложение «client-service» - будет отвечать за создание заказов на покупку или сервисное обслуживание.

Каждое приложение после создания необходимо зарегистрировать в основном файле settings.py:

```
# Applications
#####
INSTALLED_APPS += [
    'api',
    'common',
    'users',
    'clients',
    'staff',
    'equipment',
    'client_service',
]
```

Формирование базы данных приложения

В данном приложении будет использоваться база данных SQLite.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

В настройках прописано, что для нашего проекта мы будем использовать базу данных SQLite, что подходит для нас, так как не предполагает большое количество запросов. Данная настройка идёт «с коробки» и менять мы её не будем.

SQLite представляет собой быструю и легкую базу данных, которая предоставляет возможность хранить данные прямо внутри нашего приложения. Отметим ряд преимуществ использования SQLite:

1. Простота - SQLite не требует отдельного сервера или настройки. База данных работает, как часть приложения;

2. Надежность: Транзакции и ACID-свойства обеспечивают надежность и целостность данных;
3. Кроссплатформенность: SQLite поддерживается на множестве платформ, включая Windows, macOS и Linux;
4. Эффективность: SQLite требуются минимальные ресурсы системы, что отлично подходит, например, для мобильных устройств.

Установка и настройка Django Rest Framework.

Что такое Django Rest Framework

Сегодня сеть интернет построена по принципу Клиент-Серверного взаимодействия. Клиент посылает запрос — Сервер ему отвечает. В случае, когда между собой общаются два Сервера, мы условно называем Клиентом того, который отправил запрос и ожидает ответ, а Сервером будет тот, кто принимает запрос и отвечает не него. Взаимодействие браузеров и веб-сайтов (первые выступают в роли Клиента, а вторые в роли Сервера) традиционно делалось при помощи технологии html-рендеринга, именно так изначально это делал Django. Чтобы получить данные с веб-сайта, браузер отправляет запрос GET к Серверу. Сервер формирует ответ в виде html-страницы и передает ее браузеру. Так Сервер передает данные браузеру, но как браузер может передать данные Серверу? В этой самой html-странице Сервер заложил все необходимые веб-формы, заполнив которые, пользователь мог бы передать свои данные обратно на сервер. Когда вы ввели свои данные в форму на сайте, браузер отправляет Серверу запрос POST, в котором содержатся ваши данные, а Сервер обрабатывает их и записывает в базу данных.

Все это отлично работало, но уже в середине нулевых такой подход перестал удовлетворять возрастающим требованиям в веб-разработке. Появлялись мобильные приложения, различные гаджеты с доступом в интернет, и для них уже не подходил стандартный способ html-рендеринга на сервере, ведь теперь

каждому клиенту нужно было отрисовать данные по-своему. Постоянно увеличивалось взаимодействие серверов друг с другом, и html-формат уже не подходил. Для всех этих задач есть другой способ обмена данными — Web API. Смысл этого способа в том, что Сервер передает Клиенту не html-страницу, а непосредственно данные, никак не влияя на то, как эти данные будут в итоге представлены. Наиболее популярными форматами для передачи данных становятся XML и JSON. Таким образом Сервер полностью избавляется от задачи отрисовки данных. Какое-то время длился переходный период, когда разработчикам веб-приложений на Сервере приходилось поддерживать оба способа одновременно: html рендерился на Сервере для браузеров, а Web API использовался для мобильных приложений и интеграции с другими серверами. Понятно, что разработчикам приложений на Сервере приходилось делать двойную работу. Но в начале десятых ситуация стала меняться в пользу Web API. Этому способствовало молниеносное развитие инструментов на языке JavaScript, а также появление различных веб-фреймворков, одним из которых и является предмет данной статьи.

Браузерные приложения быстро научились отрисовывать веб-страницы самостоятельно, получая чистые данные с Сервера. Веб-приложения на сервере научились создавать API быстро и легко. Так сформировалась четкое разделение на Backend и Frontend разработку: тех, кто поддерживает приложение на Сервере, и тех, кто делает браузерные (клиентские) приложения. А Web API стал универсальным способом общения для Сервера и всех его клиентов (браузеров, мобильных приложений, других Серверов). Конечно, это не могло не привести к развитию стандартов в общении между системами. И Клиенту, и Серверу необходимо знать каким образом общаться с друг с другом, как передавать данные, как сообщать об ошибках. Разработчики всегда могли договориться о том, как взаимодействовать их приложениям, но наличие некоего стандарта в веб-разработке позволило бы эту задачу облегчить. И вот в начале десятых таким стандартом стала концепция REST.

REST

В 2000 году Рой Филдинг написал докторскую диссертацию, где изложил концепцию REST. Там были рекомендации о том, как спроектировать Сервер, чтобы ему было удобно общаться с Клиентами. Выделю два главных принципа создания приложений в стиле REST:

- Сервер не должен ничего знать о текущем состоянии Клиента. В запросе от Клиента должна быть вся необходимая информация для обработки этого запроса Сервером.
- Каждый ресурс на Сервере должен иметь определенный Id, а также уникальный URL, по которому осуществляется доступ к этому ресурсу.

На данный момент мы можем найти фреймворк для создания приложений в стиле REST практически для каждого языка программирования, используемого в веб-разработке. Логика построения Web API на Сервере в этих фреймворках реализована одинаково.

Действия для управления данными привязаны к определенным HTTP-методам. Существует несколько стандартных действий для работы с данными — это *Create, Read, Update, Delete*. Часто их обобщают как CRUD.

- Для создания объекта используется http-метод POST
- Для чтения — http-метод GET
- Для изменения — http-метод PUT
- Для удаления — http-метод DELETE

Django REST framework - это мощный и гибкий набор инструментов для создания Web API.

Некоторые причины, по которым вы можете захотеть использовать REST framework:

- Просматриваемый API - огромный выигрыш в удобстве использования для ваших разработчиков.
- Политики аутентификации, включая пакеты для OAuth1a и OAuth2.
- Сериализация, поддерживающая как ORM, так и non-ORM источники данных.
- Настраивается все - просто используйте обычные представления на основе функций, если вам не нужны более мощные возможности.
- Обширная документация и отличная поддержка сообщества.
- Используется и пользуется доверием всемирно известных компаний, включая Mozilla, Red Hat, Heroku и Eventbrite.

Установка и настройка

```
pip install djangorestframework
```

```
pip install markdown # Markdown support for the browsable API.
```

```
pip install django-filter # Filtering support
```

После установки необходимо добавить этот фреймворк в установленные приложения и сконфигурировать его. В этом проекте это сделано следующим образом.

```
#####
# DJANGO REST FRAMEWORK
#####
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',),

    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication',
        'rest_framework.authentication.BasicAuthentication',
    ],

    'DEFAULT_PARSER_CLASSES': [
        'rest_framework.parsers.JSONParser',
        'rest_framework.parsers.FormParser',
        'rest_framework.parsers.MultiPartParser',
        'rest_framework.parsers.FileUploadParser',
    ],

    'DEFAULT_FILTER_BACKENDS': ['django_filters.rest_framework.DjangoFilterBackend'],
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
    'DEFAULT_PAGINATION_CLASS': 'common.pagination.BasePagination',
}
```

Так же в проекте я использую библиотеку генерации схем OpenAPI Drf-spectacular.

Drf-spectacular — это библиотека генерации схем OpenAPI 3 с акцентом на расширяемость, настраиваемость и генерацию клиентов.

Это рекомендуемый способ генерации и представления схем OpenAPI. Библиотека стремится извлечь как можно больше информации о схеме, предоставляя при этом декораторы и расширения для лёгкой настройки.

Ниже приведена настройка библиотеки в файле settings.py

```
#####
# DRF SPECTACULAR
#####
SPECTACULAR_SETTINGS = {
    'TITLE': 'MY CRM',
    'DESCRIPTION': 'My CRM',
    'VERSION': '1.0.0',

    'SERVE_PERMISSIONS': [
        'rest_framework.permissions.IsAuthenticated',
    ],

    'SERVE_AUTHENTICATION': [
        'rest_framework.authentication.BasicAuthentication',
    ],

    'SWAGGER_UI_SETTINGS': {
        'deepLinking': True,
        "displayOperationId": True,
        "syntaxHighlight.active": True,
        "syntaxHighlight.theme": "arta",
        "defaultModelsExpandDepth": -1,
        "displayRequestDuration": True,
        "filter": True,
        "requestSnippetsEnabled": True,
    },

    'COMPONENT_SPLIT_REQUEST': True,
    'SORT_OPERATIONS': False,

    'ENABLE_DJANGO_DEPLOY_CHECK': False,
    'DISABLE_ERRORS_AND_WARNINGS': True,
}
```

Так же в приложении «ari» в файле «spectacular/urls» необходимо определить следующие настройки:

```
from django.urls import path

from drf_spectacular.views import SpectacularSwaggerView

urlpatterns = [
    path('', SpectacularSwaggerView.as_view(url_name='schema'), name='swagger-ui'),
]
```


Настройка административной панели

Django дает разработчикам возможность использовать модели для автоматической генерации административной панели. Последняя предоставляет собой полноценный интерфейс и даёт нам возможность управлять данными приложений: создавать, редактировать и удалять записи в базе данных, а также производить множество иных действий, управляющих работой приложений.

Административная панель даёт нам возможность быстро и удобно управлять данными и значительно упрощает жизнь разработчика в том числе и за счёт того, что является встроенным элементом и поставляется из «коробки».

С момента создания проекта административная панель уже доступна для использования по адресу <http://127.0.0.1:8000/admin/>. Но для работы с ней необходимо выполнить ряд настроек.

Для входа нам необходимо будет создать суперпользователя. При запуске проекта в базе данных зарегистрированных пользователей нет. Суперпользователь создается из консоли с помощью команды:

- `python manage.py createsuperuser` Далее нам будет предложено системой ввести логин, email и пароль для нашего администратора. После это по введенным данным мы сможем зайти в административную панель.

На старте админ-панель также будет пуста. Мы заполним её информацией в процессе разработки проекта.

Отметим, что разработчики фреймворка Django не просто с коробки добавили в него административную панель, но заложили возможность настраивать интерфейс под требования администратора сайта.

Кроме того, мы можем добавлять различные группы пользователей с разным уровнем доступа, в зависимости от стоящих перед ними задач.

Например, менеджеру, обрабатывающему заказы, нет необходимости давать доступ к созданию и удалению пользователей, товаров или категорий. Мы можем открыть для него таблицу заказов на просмотр и редактирование. Это не только позволит избежать ошибок в работе сайта из-за неумелого обращения, но и облегчит процесс обучения персонала более ограниченному интерфейсу.

Кастомизация панели администрирования.

В своем проекте я немного кастомизировала панель администрирования под свои нужды.

```
class ProfileAdmin(admin.StackedInline):
    model = Profile
    fields = ('telegram_id',)

@admin.register(User)
class UserAdmin(UserAdmin):
    change_user_password_template = None
    fieldsets = (
        (None, {'fields': ('phone_number', 'email', 'username')}),
        (_('Личная информация'),
         {'fields': ('first_name', 'last_name')}),
        (_('Permissions'), {
            'fields': ('is_active', 'is_staff', 'is_superuser', 'groups', 'user_permissions'),
        }),
        (_('Important dates'), {'fields': ('last_login',)}),
    )
    add_fieldsets = (
        (None, {
            'classes': ('wide',),
            'fields': ('email', 'phone_number', 'password1', 'password2',),
        }),
    )
    list_display = ('id', 'full_name', 'email', 'phone_number', )

    list_display_links = ('id', 'full_name',)
    list_filter = ('is_staff', 'is_superuser', 'is_active', 'groups')
    search_fields = ('first_name', 'last_name', 'id', 'email', 'phone_number',)
    ordering = ('-id',)
    filter_horizontal = ('groups', 'user_permissions',)
    readonly_fields = ('last_login',)
    inlines = (ProfileAdmin,)
```

Описание структуры и функций приложений проекта

1. Приложение «user»

Модели

Я создала 2 модели для пользователей:

-модель самого пользователя для регистрации, авторизации и аутентификации. Эта модель наследуется от модели Django AbstractUser.

```
class User(AbstractUser):
    username = models.CharField(
        *args: 'Nickname', max_length=64, unique=True, null=True, blank=True
    )
    email = models.EmailField(*args: 'Email', unique=True, null=True, blank=True)
    phone_number = PhoneNumberField(*args: 'Phone number', unique=True, null=True, blank=True)
    USERNAME_FIELD = 'username'
    REQUIRED_FIELDS = ['email']

    objects = CustomUserManager()

    class Meta:
        verbose_name = 'User'
        verbose_name_plural = 'Users'

    1 usage
    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'

    def __str__(self):
        return f'{self.full_name} ({self.pk})'

Explain | Test | Document | Fix | Ask

@receiver(post_save, sender=User)
def post_save_user(sender, instance, created, **kwargs):
    if not hasattr(instance, 'profile'):
        Profile.objects.create(user=instance)
```

-модель профиля пользователя

```
from django.db import models

Explain | Test | Document | Fix | Ask
7 usages
class Profile(models.Model):
    user = models.OneToOneField(
        to: 'users.User', models.CASCADE, related_name='profile',
        verbose_name='User', primary_key=True,
    )
    telegram_id = models.CharField(
        *args: 'Telegram ID', max_length=20, null=True, blank=True
    )

    class Meta:
        verbose_name = 'User profile'
        verbose_name_plural = 'Users profile'

    def __str__(self):
        return f'{self.user} ({self.pk})'
```

Сериализаторы

Для каждой модели проекта я создавала сериализаторы.

Сериализатор для регистрации пользователя

```

class RegistrationSerializer(serializers.ModelSerializer):
    email = serializers.EmailField()
    password = serializers.CharField(
        style={'input_type': 'password'}, write_only=True
    )
    Explain | Test | Document | Fix | Ask

    class Meta:
        model = User
        fields = (
            'id',
            'first_name',
            'last_name',
            'email',
            'password',
        )
        Explain | Test | Document | Fix | Ask

    def validate_email(self, value):
        email = value.lower()
        if User.objects.filter(email=email).exists():
            raise ParseError(
                'Пользователь с такой почтой уже зарегистрирован.'
            )
        return email

    def validate_password(self, value):
        validate_password(value)
        return value

    def create(self, validated_data):
        user = User.objects.create_user(**validated_data)
        return user

```

```

class ChangePasswordSerializer(serializers.ModelSerializer):
    old_password = serializers.CharField(write_only=True)
    new_password = serializers.CharField(write_only=True)

    class Meta:
        model = User
        fields = ('old_password', 'new_password')

    def validate(self, attrs):
        user = self.instance
        old_password = attrs.pop('old_password')
        if not user.check_password(old_password):
            raise ParseError(
                'Проверьте правильность текущего пароля.'
            )
        return attrs

    def validate_new_password(self, value):
        validate_password(value)
        return value

    def update(self, instance, validated_data):
        password = validated_data.pop('new_password')
        instance.set_password(password)
        instance.save()
        return instance

```

Сериализатор для представления данных пользователя.

```

class MeSerializer(serializers.ModelSerializer):
    profile = ProfileShortSerializer()

    class Meta:
        model = User
        fields = (
            'id',
            'first_name',
            'last_name',
            'email',
            'phone_number',
            'username',
            'profile',
            'date_joined',
        )

```

Сериализатор для поиска пользователя

```

class UserSearchListSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = (
            'id',
            'username',
            'email',
            'full_name',
        )

```

Сериализатор обновления данных пользователя.

```

class MeUpdateSerializer(serializers.ModelSerializer):
    profile = ProfileUpdateSerializer()

    Explain | Test | Document | Fix | Ask

    class Meta:
        model = User
        fields = (
            'id',
            'first_name',
            'last_name',
            'email',
            'phone_number',
            'username',
            'profile',
        )
        Explain | Test | Document | Fix | Ask

    def update(self, instance, validated_data):
        # Проверка наличия профиля
        profile_data = validated_data.pop('profile') if 'profile' in validated_data else None

        with transaction.atomic():
            instance = super().update(instance, validated_data)
            # # Update профиля
            if profile_data:
                self._update_profile(instance.profile, profile_data)
        return instance
        Explain | Test | Document | Fix | Ask
1 usage

    def _update_profile(self, profile, data):
        profile_serializer = ProfileUpdateSerializer(
            instance=profile, data=data, partial=True
        )
        profile_serializer.is_valid(raise_exception=True)
        profile_serializer.save()

```

Представления данных модели

Используя сериализаторы я создала несколько API представлений для пользователя.

Регистрация, изменение пароля и тд.

```
User = get_user_model()

Explain | Test | Document | Fix | Ask
1 usage
@extend_schema_view(
    post=extend_schema(summary='User registration', tags=['Authentication & Authorisation']),
)

class RegistrationView(generics.CreateAPIView):
    queryset = User.objects.all()
    permission_classes = [AllowAny]
    serializer_class = user_s.RegistrationSerializer

Explain | Test | Document | Fix | Ask
1 usage
@extend_schema_view(
    post=extend_schema(
        request=user_s.ChangePasswordSerializer,
        summary='Password change', tags=['Authentication & Authorisation']),
)

class ChangePasswordView(APIView):

    Explain | Test | Document | Fix | Ask

    def post(self, request):
        user = request.user
        serializer = user_s.ChangePasswordSerializer(
            instance=user, data=request.data
        )
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(status=HTTP_204_NO_CONTENT)
```

```

@extend_schema_view(
    get=extend_schema(summary='User profile', tags=['Users']),
    put=extend_schema(summary='Change user profile', tags=['Users']),
    patch=extend_schema(summary='Change user profile partialy', tags=['Users']),
)
class MeView(RetrieveUpdateAPIView):
    queryset = User.objects.all()
    serializer_class = user_s.MeSerializer
    http_method_names = ('get', 'patch')

    Explain | Test | Document | Fix | Ask

    def get_serializer_class(self):
        if self.request.method in ['PUT', 'PATCH']:
            return user_s.MeUpdateSerializer
        return user_s.MeSerializer

    def get_object(self):
        return self.request.user

    Explain | Test | Document | Fix | Ask
1 usage
@extend_schema_view(
    list=extend_schema(summary='Users list search', tags=['Dicts']),
)
class UserListSearchView(ListViewSet):
    queryset = User.objects.exclude(
        Q(is_superuser=True))

    serializer_class = user_s.UserSearchListSerializer
    filter_backends = (
        SearchFilter,
    )
    search_fields = ('last_name', 'email', 'username',)

```

2. Приложение «staff»

Модели

В данном приложении были созданы 2 модели данных:

- модель «Manager»

Она связана с моделью данных «User»

```
class Manager(models.Model):
    LOW = 'low'
    HIGH = 'high'
    CHOICES_LEVEL = (
        (LOW, 'Low'),
        (HIGH, 'High'),
    )

    SALES = 'sales'
    WARRANTY = 'warranty'
    CHOICES_TYPE = (
        (SALES, 'Sales'),
        (WARRANTY, 'Warranty'),
    )

    name = models.OneToOneField(User, related_name='manager', on_delete=models.CASCADE)
    level = models.CharField(max_length=10, choices=CHOICES_LEVEL, default=LOW)
    manage_role = models.CharField(max_length=20, choices=CHOICES_TYPE, default=None)
    created_at = models.DateTimeField(auto_now_add=True)
    modified_at = models.DateTimeField(auto_now=True)
    is_active = models.BooleanField(default=True)

    def __str__(self):
        return f'{self.name} --- {self.manage_role}'
```

- модель данных «Employee», которая так же связана с «User»

```

class Employee(models.Model):
    FIRST_CATEGORY = 'first_category'
    SECOND_CATEGORY = 'second_category'
    THIRD_CATEGORY = 'third_category'
    LEAD_ENGINEER = 'lead_engineer'

    CHOICES_CATEGORY = (
        (FIRST_CATEGORY, 'First Category'),
        (SECOND_CATEGORY, 'Second Category'),
        (THIRD_CATEGORY, 'Third Category'),
        (LEAD_ENGINEER, 'Lead Engineer'),
    )

    ELECTRONICS = 'electronics'
    MECHANICS = 'mechanics'
    UNIVERSAL = 'universal'

    CHOICES_ROLE = (
        (ELECTRONICS, 'Electronics'),
        (MECHANICS, 'Mechanics'),
        (UNIVERSAL, 'Universal'),
    )

    name = models.OneToOneField(User, related_name='employee', on_delete=models.CASCADE)
    category = models.CharField(max_length=20, choices=CHOICES_CATEGORY, default=FIRST_CATEGORY)
    role = models.CharField(max_length=20, choices=CHOICES_ROLE, default=UNIVERSAL)
    created_at = models.DateTimeField(auto_now_add=True)
    modified_at = models.DateTimeField(auto_now=True)
    is_active = models.BooleanField(default=True)

    def __str__(self):
        return f'{self.name} --- {self.category}'


```

Обе модели содержат в себе данные на сотрудников. Часть является менеджерами, часть — сервисными работниками. Я использовала атрибут `choices`, для выбора типа, уровня и т.д. для сотрудников.

Сериализаторы

- для модели «Manager»

```
class ManagerSerializer(ExtendedModelSerializer):  
    user = UserShortSerializer()  
  
    class Meta:  
        model = Manager  
        fields = ('user', 'level', 'manage_role')
```

 Explain | Test | Document | Fix | Ask

2 usages

```
class ManagerSearchSerializer(ExtendedModelSerializer):  
    user = UserShortSerializer()  
  
    class Meta:  
        model = Manager  
        fields = ('user', 'level', 'manage_role')
```

```

class ManagerCreateSerializer(ExtendedModelSerializer):
    first_name = serializers.CharField(write_only=True)
    last_name = serializers.CharField(write_only=True)
    email = serializers.EmailField(write_only=True)
    password = serializers.CharField(write_only=True)

    Explain | Test | Document | Fix | Ask

class Meta:
    model = Manager
    fields = (
        'id',
        'first_name',
        'last_name',
        'email',
        'password',
        'level',
        'manage_role',
    )

    Explain | Test | Document | Fix | Ask

def create(self, validated_data):
    user_data = {
        'first_name': validated_data.pop('first_name'),
        'last_name': validated_data.pop('last_name'),
        'email': validated_data.pop('email'),
        'password': validated_data.pop('password'),
    }

    with transaction.atomic():
        user = User.objects.create_user(**user_data)
        validated_data['user'] = user

    instance = super().create(validated_data)

```

```
class ManagerListSerializer(ExtendedModelSerializer):
    user = UserShortSerializer()

    class Meta:
        model = Manager
        fields = ('user', 'level', 'manage_role')
```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

2 usages

```
class ManagerUpdateSerializer(ExtendedModelSerializer):
```

```
    class Meta:
        model = Manager
        fields = ('level', 'manage_role')
```


[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

```
def update(self, instance, validated_data):
    category_data = validated_data.pop('level', instance.category)
    role_data = validated_data.pop('manage_role', instance.role)

    instance = super().update(instance, validated_data)
    instance.category.set(*[category_data])
    instance.role.set(*[role_data])
```

- для модели «Employee»

```
class EmployeeSerializer(ExtendedModelSerializer):  
    user = UserShortSerializer()  
  
    class Meta:  
        model = Employee  
        fields = ('user', 'category', 'role')
```

 Explain | Test | Document | Fix | Ask

2 usages

```
class EmployeeSearchSerializer(ExtendedModelSerializer):  
    user = UserShortSerializer()  
  
    class Meta:  
        model = Employee  
        fields = ('user', 'category', 'role')
```



```

class EmployeeCreateSerializer(ExtendedModelSerializer):
    first_name = serializers.CharField(write_only=True)
    last_name = serializers.CharField(write_only=True)
    email = serializers.EmailField(write_only=True)
    password = serializers.CharField(write_only=True)

    Explain | Test | Document | Fix | Ask

    class Meta:
        model = Employee
        fields = (
            'id',
            'first_name',
            'last_name',
            'email',
            'password',
        )

    Explain | Test | Document | Fix | Ask

    def create(self, validated_data):
        user_data = {
            'first_name': validated_data.pop('first_name'),
            'last_name': validated_data.pop('last_name'),
            'email': validated_data.pop('email'),
            'password': validated_data.pop('password'),
        }

        with transaction.atomic():
            user = User.objects.create(**user_data)
            validated_data['user'] = user

            instance = super().create(validated_data)
        return instance

```

```

class EmployeeListSerializer(ExtendedModelSerializer):
    user = UserShortSerializer()

    class Meta:
        model = Employee
        fields = ('user', 'category', 'role')

```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

2 usages

```

class EmployeeUpdateSerializer(ExtendedModelSerializer):

    class Meta:
        model = Employee
        fields = ('category', 'role')

```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

```

def update(self, instance, validated_data):
    category_data = validated_data.pop('category', instance.category)
    role_data = validated_data.pop('role', instance.role)

    instance = super().update(instance, validated_data)
    instance.category.set(*[category_data])
    instance.role.set(*[role_data])

```

Представления данных модели

- для данных модели «Manager»

```
@extend_schema_view(
    list=extend_schema(summary='List of managers', tags=['Company:']),
    create=extend_schema(summary='Create managers', tags=['Company:']),
    update=extend_schema(summary='Change managers', tags=['Company:']),
    partial_update=extend_schema(summary='Change managers partially', tags=['Company:']),
    destroy=extend_schema(summary='Delete managers', tags=['Company:']),
    search=extend_schema(filters=True, summary='Search managers', tags=['Dicts']),
)

class ManagerView(LCRUDViewSet):
    queryset = Manager.objects.all()
    serializer_class = managers_s.ManagerListSerializer

    multi_serializer_class = {
        'list': managers_s.ManagerListSerializer,
        'create': managers_s.ManagerCreateSerializer,
        'update': managers_s.ManagerUpdateSerializer,
        'partial_update': managers_s.ManagerUpdateSerializer,
        'search': managers_s.ManagerSearchSerializer,
        'destroy': managers_s.ManagerSearchSerializer,
    }

    lookup_url_kwarg = 'user_id'
    http_method_names = ('get', 'post', 'patch', 'delete',)

    filter_backends = (
        DjangoFilterBackend,
        OrderingFilter,
        SearchFilter,
    )
```

```

def get_queryset(self):
    qs = Manager.objects.select_related(
        'user',
        'level',
        'manage_role',
    )
    return qs

@action(methods=['GET'], detail=False, url_path='search')
def search(self, request, *args, **kwargs):
    return super().list(request, *args, *args, **kwargs)

Explain | Test | Document | Fix | Ask

def destroy(self, request, *args, **kwargs):
    instance = self.get_object()
    serializer = self.get_serializer(*args: instance, data=dict())
    serializer.is_valid(raise_exception=True)
    return super().destroy(request, *args: *args, **kwargs)

```

- Для данных модели «Employee»

```

@extend_schema_view(
    list=extend_schema(summary='List of engineers', tags=['Company:']),
    create=extend_schema(summary='Create engineer', tags=['Company:']),
    update=extend_schema(summary='Change engineer', tags=['Company:']),
    partial_update=extend_schema(summary='Change engineer partially', tags=['Company:']),
    destroy=extend_schema(summary='Delete engineer', tags=['Company:']),
    search=extend_schema(filters=True, summary='Search engineer', tags=['Dicts']),
)

class EmployeeView(LCRUDViewSet):
    queryset = Employee.objects.all()
    serializer_class = employees_s.EmployeeListSerializer

    multi_serializer_class = {
        'list': employees_s.EmployeeListSerializer,
        'create': employees_s.EmployeeCreateSerializer,
        'update': employees_s.EmployeeUpdateSerializer,
        'partial_update': employees_s.EmployeeUpdateSerializer,
        'search': employees_s.EmployeeSearchSerializer,
        'destroy': employees_s.EmployeeSearchSerializer,
    }

    lookup_url_kwarg = 'user_id'
    http_method_names = ('get', 'post', 'patch', 'delete',)

    filter_backends = (
        DjangoFilterBackend,
        OrderingFilter,
        SearchFilter,
    )

```

```

def get_queryset(self):
    qs = Employee.objects.select_related(
        'user',
        'category',
        'role',
    )
    return qs

    .

@action(methods=['GET'], detail=False, url_path='search')
def search(self, request, *args, **kwargs):
    return super().list(request, *args: *args, **kwargs)

    Explain | Test | Document | Fix | Ask

def destroy(self, request, *args, **kwargs):
    instance = self.get_object()
    serializer = self.get_serializer(*args: instance, data=dict())
    serializer.is_valid(raise_exception=True)
    return super().destroy(request, *args: *args, **kwargs)

```

3. Приложение «client-service»

В этом приложении модели имеют множественные связи с другими моделями. Они включают в себя менеджеров, клиентов, оборудование или работы, которые необходимо выполнить.

Это приложение формирует заказы на приобретение товаров или услуг.

Рассмотрим модели данного приложения

Модели

- модель данных «Sales_order»

```
class SalesOrder(InfoMixin):
    order_code = models.CharField(max_length=20, blank=False, null=False, primary_key=True)
    customer = models.ForeignKey(Client, related_name='sales_order', on_delete=models.CASCADE)
    manager = models.ForeignKey(Manager, related_name='sales_order', on_delete=models.PROTECT)
    equipment = models.ForeignKey(Equipment, related_name='sales_order', on_delete=models.CASCADE)
    quantity = models.PositiveIntegerField(blank=False, null=False)
    price = models.PositiveIntegerField(blank=True, null=True, default=0)
    # created_by = models.ForeignKey(User, related_name='sales_order', on_delete=models.PROTECT)
    order_date = models.DateTimeField(auto_now_add=True)

    # modified_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f'{self.order_code} / {self.customer}'
```

- Модель данных «Service_order»

```

class ServiceOrder(InfoMixin):
    MAINTENANCE = 'maintenance'
    COMMISSIONING = 'commissioning'
    CHOICES_TYPE = (
        (MAINTENANCE, 'Maintenance'),
        (COMMISSIONING, 'Commissioning'),
    )

    HIGH = 'high'
    MEDIUM = 'medium'
    LOW = 'low'
    CHOICES_DIFFICULTY = (
        (HIGH, 'High'),
        (MEDIUM, 'Medium'),
        (LOW, 'Low'),
    )

    order_code = models.CharField(max_length=20, blank=False, null=False, primary_key=True)
    customer = models.ForeignKey(Client, related_name='service_order', on_delete=models.CASCADE)
    manager = models.ForeignKey(Manager, related_name='service_order', on_delete=models.PROTECT)
    engineer = models.ForeignKey(Employee, related_name='service_order', on_delete=models.PROTECT)
    equipment = models.ForeignKey(Equipment, related_name='service_order', on_delete=models.PROTECT)
    type = models.CharField(max_length=20, choices=CHOICES_TYPE, blank=False, null=False)
    difficulty = models.CharField(max_length=20, choices=CHOICES_DIFFICULTY, blank=False, null=False)
    # created_by = models.ForeignKey(User, related_name='service_order', on_delete=models.PROTECT)
    order_date = models.DateTimeField(auto_now_add=True)

    # modified_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f'{self.order_code} / {self.customer}'

```

Сериализаторы

- для данных модели «Sales_order»


```
class SalesSearchSerializer(ExtendedModelSerializer):
```

```
    class Meta:
```

```
        model = SalesOrder
```

```
        fields = ('order_code', 'customer', 'order_date')
```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

1 usage

```
class SalesCreateSerializer(ExtendedModelSerializer):
```

```
    customer = ClientSerializer(read_only=True)
```

```
    manager = ManagerSerializer(read_only=True)
```

```
    equipment = EquipmentSerializer(many=True, read_only=True)
```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

```
    class Meta:
```

```
        model = SalesOrder
```

```
        fields = (
```

```
            'order_code',
```

```
            'customer',
```

```
            'manager',
```

```
            'equipment',
```

```
            'quantity',
```

```
            'price',
```

```
        )
```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

```
    def create(self, validated_data):
```

```
        customer_data = validated_data.pop("customer")
```

```
        manager_data = validated_data.pop("manager")
```

```
        equipment_data = validated_data.pop("equipment")
```


```
        customer_obj = Client.objects.create(**customer_data)  # Learn more
```

```
class SalesListSerializer(ExtendedModelSerializer):
```

```
    class Meta:
```

```
        model = SalesOrder
```

```
        fields = ('__all__',)
```

 Explain | Test | Document | Fix | Ask


2 usages

```
class SalesUpdateSerializer(ExtendedModelSerializer):
```

```
    class Meta:
```

```
        model = SalesOrder
```

```
        fields = ('quantity', 'price')
```

 Explain | Test | Document | Fix | Ask

```
    def update(self, instance, validated_data):
```

```
        quantity_data = validated_data.pop('quantity', instance.quantity)
```

```
        price_data = validated_data.pop('price', instance.price)
```

```
        instance = super().update(instance, validated_data)
```

```
        instance.description.set(*[quantity_data])
```

```
        instance.price.set(*[price_data])
```

- для данных модели «Service_order»

```
class ServiceSearchSerializer(ExtendedModelSerializer):
```

```
    class Meta:
```

```
        model = ServiceOrder
```

```
        fields = ('order_code', 'customer', 'order_date')
```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

1 usage

```
class ServiceCreateSerializer(ExtendedModelSerializer):
```

```
    customer = ClientSerializer(read_only=True)
```

```
    manager = ManagerSerializer(read_only=True)
```

```
    engineer = EmployeeSerializer(many=True, read_only=True)
```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

```
    class Meta:
```

```
        model = ServiceOrder
```

```
        fields = (
```

```
            'order_code',
```

```
            'customer',
```

```
            'manager',
```

```
            'engineer',
```

```
            'type',
```

```
            'difficulty',
```

```
        )
```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

```
    def create(self, validated_data):
```

```
        customer_data = validated_data.pop("customer")
```

letSerializer = Meta

```

class ServiceListSerializer(ExtendedModelSerializer):

    class Meta:
        model = ServiceOrder
        fields = ('__all__')

Explain | Test | Document | Fix | Ask
2 usages

class ServiceUpdateSerializer(ExtendedModelSerializer):

    class Meta:
        model = ServiceOrder
        fields = ('type', 'difficulty')

Explain | Test | Document | Fix | Ask

    def update(self, instance, validated_data):
        description_data = validated_data.pop('type', instance.description)
        price_data = validated_data.pop('difficulty', instance.price)

        instance = super().update(instance, validated_data)
        instance.description.set(*[description_data])
        instance.price.set(*[price_data])

```

Представления

На базе моделей и сериализаторов были созданы представления этих данных.

- для данных «Sales_order»

```

@extend_schema_view(
    list=extend_schema(summary='List of Sales', tags=['Sales']),
    create=extend_schema(summary='Create Sales', tags=['Sales']),
    update=extend_schema(summary='Change Sales', tags=['Sales']),
    partial_update=extend_schema(summary='Change Sales partialy', tags=['Sales']),
    destroy=extend_schema(summary='Delete Sales', tags=['Sales']),
    search=extend_schema(filters=True, summary='Search Sales', tags=['Dicts']),
)

class SalesView(LCRUDViewSet):
    queryset = SalesOrder.objects.all()
    serializer_class = sales_order_s.SalesListSerializer

    multi_serializer_class = {
        'list': sales_order_s.SalesListSerializer,
        'create': sales_order_s.SalesCreateSerializer,
        'update': sales_order_s.SalesUpdateSerializer,
        'partial_update': sales_order_s.SalesUpdateSerializer,
        'search': sales_order_s.SalesSearchSerializer,
        'destroy': sales_order_s.SalesSearchSerializer,
    }

    http_method_names = ('get', 'post', 'patch', 'delete',)

    filter_backends = (
        DjangoFilterBackend,
        OrderingFilter,
        SearchFilter,
    )

```

```

def get_queryset(self):
    qs = SalesOrder.objects.select_related(
        'order_code',
        'customer',
        'manager',
        'equipment',
        'quantity',
        'price',
    )
    return qs

@action(methods=['GET'], detail=False, url_path='search')
def search(self, request, *args, **kwargs):
    return super().list(request, *args: *args, **kwargs)

def destroy(self, request, *args, **kwargs):
    instance = self.get_object()
    serializer = self.get_serializer(*args: instance, data=dict())
    serializer.is_valid(raise_exception=True)
    return super().destroy(request, *args: *args, **kwargs)

```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

- для данных «Service_order»

```

@extend_schema_view(
    list=extend_schema(summary='List of Service', tags=['Service']),
    create=extend_schema(summary='Create Service', tags=['Service']),
    update=extend_schema(summary='Change Service', tags=['Service']),
    partial_update=extend_schema(summary='Change Service partialy', tags=['Se
    destroy=extend_schema(summary='Delete Service', tags=['Service']),
    search=extend_schema(filters=True, summary='Search Sales', tags=['Dicts'])
)

class ServiceView(LCRUDViewSet):
    queryset = ServiceOrder.objects.all()
    serializer_class = service_order_s.ServiceListSerializer

    multi_serializer_class = {
        'list': service_order_s.ServiceListSerializer,
        'create': service_order_s.ServiceCreateSerializer,
        'update': service_order_s.ServiceUpdateSerializer,
        'partial_update': service_order_s.ServiceUpdateSerializer,
        'search': service_order_s.ServiceSearchSerializer,
        'destroy': service_order_s.ServiceSearchSerializer,
    }

    http_method_names = ('get', 'post', 'patch', 'delete',)

    filter_backends = (
        DjangoFilterBackend,
        OrderingFilter,
        SearchFilter,
    )

```

```

def get_queryset(self):
    qs = ServiceOrder.objects.select_related(
        'order_code',
        'customer',
        'manager',
        'equipment',
        'type',
        'difficulty',
    )
    return qs

@action(methods=['GET'], detail=False, url_path='search')
def search(self, request, *args, **kwargs):
    return super().list(request, *args: *args, **kwargs)

def destroy(self, request, *args, **kwargs):
    instance = self.get_object()
    serializer = self.get_serializer(*args: instance, data=dict())
    serializer.is_valid(raise_exception=True)
    return super().destroy(request, *args: *args, **kwargs)

```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

4. Приложение «Common»

Миксины для моделей

В этом приложении собраны некоторые классы и функции которые используются в проекте в целом, вне зависимости от приложения.

Для упрощения создания моделей я использовала миксины.

В программировании можно столкнуться с ситуациями, когда нужно расширить функциональность классов без создания сложных иерархий наследования. Миксины в Python являются одним из способов решения этой проблемы.

В этом уроке мы рассмотрим, что такое миксины, как они используются, и как можно с их помощью упростить структуру кода.

Понятие миксинов

Миксины или **Mixins** — это форма множественного наследования в Python и мощный инструмент, который позволяет преодолеть ограничения единственного наследования. Они представляют собой простые классы, которые включают набор методов, предназначенных для добавления к другому классу, и позволяют расширять функциональность классов без глубокой иерархии наследования.

Это устраняет проблемы, связанные с множественным наследованием, и делает миксины гибким средством для улучшения и модификации структуры кода.

Миксины создаются для того, чтобы повторно использовать функции во множестве классов. Они не предполагают создание объектов, и не должны иметь своего состояния.

```

class DateMixin(models.Model):
    created_at = models.DateTimeField(verbose_name='Created at', null=True, blank=False)
    updated_at = models.DateTimeField(verbose_name='Updated at', null=True, blank=False)

    class Meta:
        abstract = True

    def save(self, *args, **kwargs):
        if not self.pk and not self.created_at:
            self.created_at = timezone.now()
        self.updated_at = timezone.now()
        return super(DateMixin, self).save(*args, **kwargs)

```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

```

class InfoMixin(DateMixin):
    created_by = models.ForeignKey(
        User, models.SET_NULL, related_name='created_%(app_label)s_%(class)s',
        verbose_name='Created by', null=True, )
    updated_by = models.ForeignKey(
        User, models.SET_NULL, related_name='updated_%(app_label)s_%(class)s',
        verbose_name='Updated by', null=True, )

    class Meta:
        abstract = True

    def save(self, *args, **kwargs):
        from crum import get_current_user

        user = get_current_user()
        if user and not user.pk:
            user = None
        if not self.pk:
            self.created_by = user
        self.updated_by = user
        super().save(*args, *args, **kwargs)

```

[Explain](#) | [Test](#) | [Document](#) | [Fix](#) | [Ask](#)

Миксины для сериализаторов

```
class ExtendedModelSerializer(serializers.ModelSerializer):  
  
    class Meta:  
        abstract = True  
  
    def get_from_url(self, lookup_field):  
        assert 'view' in self.context, (  
            'No view context in "%s". ' % self.__class__.__name__  
            'Check parameter context on function calling.'  
        )  
        assert self.context['view'].kwargs.get(lookup_field), (  
            'Got no data from url in "%s". ' % self.__class__.__name__  
            'Check lookup field on function calling.'  
        )  
        value = self.context['view'].kwargs.get(lookup_field)  
        return value  
  
    def get_object_from_url(self, model, lookup_field='pk', model_field='pk'):  
        obj_id = self.get_from_url(lookup_field)  
        obj = get_object_or_404(  
            queryset=model.objects.all(), **{model_field: obj_id}  
        )  
        return obj
```

5. URLs

Файл `urls.py` в Django является основным механизмом для маршрутизации URL-адресов приложения. Он используется для соответствия URL-адресов определенным функциям (вызываемым `view`-функциям) в вашем приложении Django.

Обычно в Django проектах есть два вида файлов `urls.py` — один для основного проекта и один для каждого приложения в проекте. Файл `urls.py` приложения определяет URL-адреса, специфичные для данного приложения, а файл `urls.py` проекта используется для обработки URL-адресов, которые не определены в `urls.py` приложения.

`urlpatterns` — это список, содержащий все пути маршрутизации URL-адресов для приложения Django. Он обрабатывается в том порядке, в котором описаны URL-адреса, сверху вниз, до тех пор, пока не будет найден соответствующий маршрут для запроса.

Каждый элемент списка `urlpatterns` обычно содержит объект `path()` или `re_path()`, который определяет маршрут URL-адреса, а также имя вызываемой `view`-функции. Также в этом списке можно использовать специальные функции, такие как `include()`, которая позволяет добавлять в `urlpatterns` другие списки `urlpatterns` из других модулей.

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

from client_service.views import sales_view, service_view

router = DefaultRouter()

router.register(prefix='sales_view', sales_view.SalesView, basename='sales_view')
router.register(prefix='service_view', service_view.ServiceView, basename='service_view')

urlpatterns = [
    path('client_service/', include(router.urls)),
]
```

В данном приложении я использую списки `urlpatterns` и объединяю их в одном файле приложения «api».

```
from api.spectacular.urls import urlpatterns as doc_urls
from users.urls import urlpatterns as user_urls
from django.urls import path, include
from staff.urls import urlpatterns as staff_urls
from equipment.urls import urlpatterns as equipment_urls
from clients.urls import urlpatterns as clients_urls
from client_service.urls import urlpatterns as client_service_urls
app_name = 'api'

urlpatterns = [
    path('auth/', include('djoser.urls.jwt')),
]

urlpatterns += doc_urls
urlpatterns += user_urls
urlpatterns += staff_urls
urlpatterns += equipment_urls
urlpatterns += clients_urls
urlpatterns += client_service_urls
```

6. Пагинация

DRF включает поддержку настраиваемых стилей пагинации. Это позволяет изменять, как большие наборы результатов разбиваются на отдельные страницы данных.

API пагинации может поддерживать любую из этих функций:

- Ссылки пагинации, которые предоставляются как часть содержимого ответа.
- Ссылки пагинации, включенные в заголовки ответа, такие как `Content-Range` или `Link`.

В настоящее время все встроенные стили используют ссылки, включенные как часть содержимого ответа. Этот стиль более доступен при использовании API с возможностью просмотра.

Пагинация выполняется автоматически, только если вы используете общие представления или наборы представлений. Если вы используете обычное APIView, вам нужно будет самостоятельно обратиться к API пагинации, чтобы убедиться, что вы возвращаете ответ с пагинацией. Пример смотрите в исходном коде классов mixins.ListModelMixin и generics.GenericAPIView.

В приложении «common» я создал файл pagination.py со следующим содержимым:

```
from rest_framework.pagination import PageNumberPagination
from rest_framework.response import Response

class BasePagination(PageNumberPagination):
    page_size_query_param = 'page_size'
    max_page_size = 1000

    def get_paginated_response(self, data):
        return Response({
            'next': self.get_next_link(),
            'previous': self.get_previous_link(),
            'count': self.page.paginator.count,
            'pages': self.page.paginator.num_pages,
            'results': data
        })
```

7. Итоговое представление REST API

MY CRM

1.0.0 OAS 3.0

/schema/

My CRM

Authorize

Filter by tag

api

POST

/api/auth/jwt/create/

api_auth_jwt_create_create

POST

/api/auth/jwt/refresh/

api_auth_jwt_refresh_create

POST

/api/auth/jwt/verify/

api_auth_jwt_verify_create

GET

/api/company/employees/{user_id}/

api_company_employees_retrieve

GET

/api/company/managers/{user_id}/

api_company_managers_retrieve

GET

/api/equipment/equipment/{equipment_id}/

api_equipment_equipment_retrieve

Pull Requests	/api/client_service/service_view/search/ Search Sales	api_client_service_service_view_search_retrieve	🔒	▼
Company:				
GET	/api/company/employees/ List of engineers	api_company_employee_s_list	🔒	▼
POST	/api/company/employees/ Create engineer	api_company_employees_create	🔒	▼
PATCH	/api/company/employees/{user_id}/ Change engineer partialy	api_company_employees_partial_update	🔒	▼
DELETE	/api/company/employees/{user_id}/ Delete engineer	api_company_employees_destroy	🔒	▼
GET	/api/company/managers/ List of managers	api_company_managers_list	🔒	▼
POST	/api/company/managers/ Create managers	api_company_managers_create	🔒	▼
PATCH	/api/company/managers/{user_id}/ Change managers partialy	api_company_managers_partial_update	🔒	▼
DELETE	/api/company/managers/{user_id}/ Delete managers	api_company_managers_destroy	🔒	▼
Equipment				
GET	/api/equipment/equipment/ List of equipment	api_equipment_equipment_list	🔒	▼
POST	/api/equipment/equipment/ Create equipment	api_equipment_equipment_create	🔒	▼

PATCH	/api/clients/clients/{clients_id}/ Change clients partialy	api_clients_clients_partial_update	🔒	▼
DELETE	/api/clients/clients/{clients_id}/ Delete clients	api_clients_clients_destroy	🔒	▼
Sales				
GET	/api/client_service/sales_view/ List of Sales	api_client_service_sales_view_list	🔒	▼
POST	/api/client_service/sales_view/ Create Sales	api_client_service_sales_view_create	🔒	▼
PATCH	/api/client_service/sales_view/{order_code}/ Change Sales partialy	api_client_service_sales_view_partial_update	🔒	▼
DELETE	/api/client_service/sales_view/{order_code}/ Delete Sales	api_client_service_sales_view_destroy	🔒	▼
Service				
GET	/api/client_service/service_view/ List of Service	api_client_service_service_view_list	🔒	▼
POST	/api/client_service/service_view/ Create Service	api_client_service_service_view_create	🔒	▼
PATCH	/api/client_service/service_view/{order_code}/ Change Service partialy	api_client_service_service_view_partial_update	🔒	▼
DELETE	/api/client_service/service_view/{order_code}/ Delete Service	api_client_service_service_view_destroy	🔒	▼
schema				
GET	/schema/	schema_retrieve	🔒	▼

Дальнейшее развитие проекта

Дальнейшее развития проекта я вижу в расширении функционала проекта.

- Создать front-end представление для данного проекта на VUE.js;
- Добавить к функционалу возможность расширенного менеджмента товарами и услугами.
- Заменить базу данных на Postgresql.
- Поработать с очередями задач используя celery
- Разработать систему логирования и сбора метрик.

Звучит просто, на деле, как всегда, потребует максимального погружения.

Django, как омут, но омут полезный, наполненный тоннами полезных фишек.

Подводя итоги данного раздела, отметим, что в какую бы сторону не стал развиваться наш проект, мы обязательно найдем решение для новой реализации.

ЗАКЛЮЧЕНИЕ

Подводя итоги проведенной работы, можно смело сказать, что фреймворк Django справедливо считается одним из самых популярных в среде разработки.

Чтобы изучить все возможности Django мало одного проекта, с ним хочется работать и дальше, изучая другие библиотеки и средства реализации различных функциональностей. Фреймворк обладает развитой экосистемой. Все более чем 18 лет существования он изменялся и совершенствовался. Какая бы задача не стояла перед нами в веб-разработке с большой вероятностью мы сможем найти ответ в уже готовом решении. Последнему немало способствует большое комьюнити разработчиков, использующих данную платформу, а также большое количество поддерживаемых библиотек.

В результате своей работы мы полностью реализовали задачи, которые ставили перед собой на старте работы над проектом:

Изучили структуру фреймворка Django;

Познакомились и применили на практике инструменты работы с изображениями, сессиями, формами, моделями, базами данных;

Получили по итогу полноценный REST API, готовый к работе с front-end.

Благодаря гибкости, которую нам даёт фреймворк проект может расти и развиваться в любую сторону. Мы можем добавить на наш сайт раздел с блогом, подключить регистрацию через социальные сети и много многое другое.

СПИСОК ЛИТЕРАТУРЫ

1. [Введение в Django & DevOps](#)
2. [Django учебник Часть 3: Использование моделей](#)
3. [AndreiMalevanyi. Работа с SQLite в Python](#)
4. [Начинаем работу с Django — подключение админки](#)
5. [Django в примерах](#)
6. [Python: Разработка на фреймворке Django](#)
7. [Формы, связанные с моделями. Пользовательские валидаторы](#)
8. [Пример Django Admin Stacked Inline: отношения многие-к-одному и многие-ко-многим](#)
9. [Django template Using Message Message Framework](#)
10. [Постраничная навигация \(пагинация\)](#)