Установите Poetry, используя pip:

pip install poetry

Инициализируйте Poetry внутри вашего проекта:

poetry init

Poetry задаст несколько вопросов, чтобы настроить ваш проект. Вы можете пропустить их, нажав Enter, чтобы использовать значения по умолчанию.

Добавьте зависимости в ваш проект, используя команду **poetry add**. Например, для установки Django:

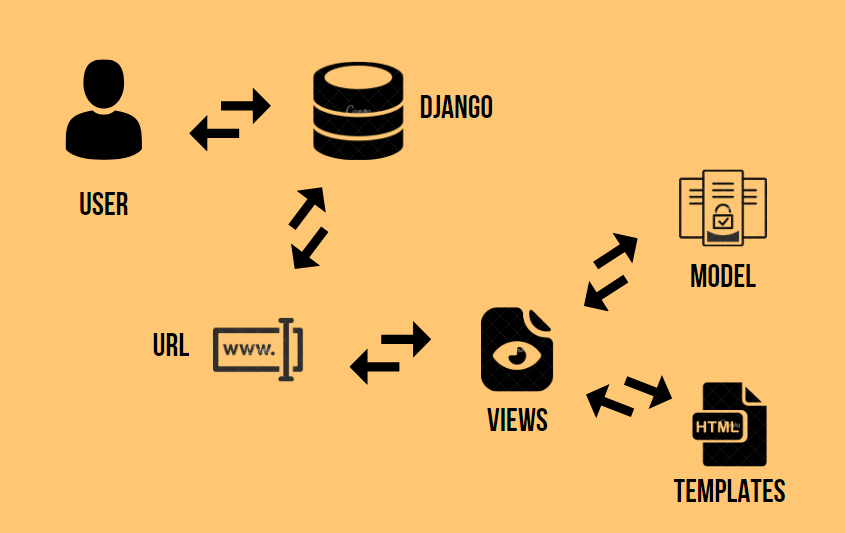
poetry add django

Установите зависимости в ваше виртуальное окружение с помощью команды **poetry install**.

После добавления новых зависимостей, запустите команду **poetry update**, чтобы обновить зависимости в вашем проекте.

Если вы хотите узнать больше о Poetry, рекомендуем ознакомиться с документацией.

Django – это фреймворк, который использует паттерн проектирования Model-View-Controller (MVC), или, точнее, Model-View-Template (MVT).



Архитектура Django приложений основана на этом паттерне и включает следующие компоненты:

1. Модели (Models): это классы, которые определяют объекты данных и их связи с базой данных. Модели обычно определяются в файле models.py.
2. Представления (Views): это функции или классы, которые обрабатывают запросы HTTP и возвращают ответы. Они используют модели и шаблоны для взаимодействия с данными и отображения информации на веб-страницах. Представления обычно определяются в файле views.py.
3. Шаблоны (Templates): это файлы HTML, которые определяют внешний вид веб-страницы. Они обычно содержат блоки, которые заполняются данными из представлений. Шаблоны обычно хранятся в директории templates.
4. URL-адреса (URLs): это файлы, которые определяют, какие представления должны быть вызваны для каждого URL-адреса. Они могут использовать регулярные выражения для обработки динамических URL-адресов. URL-адреса обычно определяются в файле urls.py.
5. Статические файлы (Static Files): это файлы, такие как изображения, стили CSS и скрипты JavaScript, которые используются для оформления веб-страниц. Они обычно хранятся в директории static.
6. Файлы конфигурации (Configuration Files): это файлы, которые используются для настройки проекта Django. Например, settings.py содержит настройки проекта, urls.py определяет глобальные URL-адреса, а wsgi.py используется для развертывания проекта на сервере.

Таким образом, архитектура Django приложений разделяет логику приложения на три отдельных компонента – модели, представления и шаблоны. Это позволяет разработчикам легко организовывать свой код, сделать его более читабельным и переиспользуемым.

Контроллер в Django называется URL-маршрутизатором, который определяет, какие обработчики запросов (View) должны быть вызваны для каждого URL-адреса. URL-маршрутизатор определяется в файле urls.py в каждом приложении Django.

В итоге, в Django API-приложениях модель (Model) определяет структуру и поведение данных, обработчик запросов (View) обрабатывает запросы и возвращает ответы, а URL-маршрутизатор определяет, какие обработчики запросов (View) должны быть вызваны для каждого URL-адреса.

Чтобы инициализировать ваше первое Django приложение в существующем каталоге с помощью poetry, нужно выполнить следующие шаги:

* Убедитесь, что у вас установлен Poetry. Если нет, установите его с помощью команды:

curl -sSL <https://raw.githubusercontent.com/python-poetry/poetry/master/get-poetry.py> | python -

* Перейдите в каталог, где вы хотите создать проект Django, и выполните команду:

poetry init

* Следуйте инструкциям в интерактивном режиме, чтобы заполнить информацию о вашем проекте, включая имя проекта и имя автора.
* Добавьте Django в ваш проект, выполнив команду:

poetry add django

* Инициализируйте проект Django, используя команду:

poetry run django-admin startproject myproject .

Замените **myproject** на желаемое имя вашего проекта. Обратите внимание на точку в конце команды, она указывает, что проект должен быть создан в текущем каталоге.

* Вот дерево структуры проекта, созданного с помощью вышеуказанных команд:

myproject\_repository/

├── project/

│ ├── \_\_init\_\_.py

│ ├── asgi.py

│ ├── settings.py

│ ├── urls.py

│ └── wsgi.py

├── manage.py

├── poetry.lock

└── pyproject.toml

* Создайте миграции базы данных для проекта, используя команду:

poetry run python manage.py makemigrations

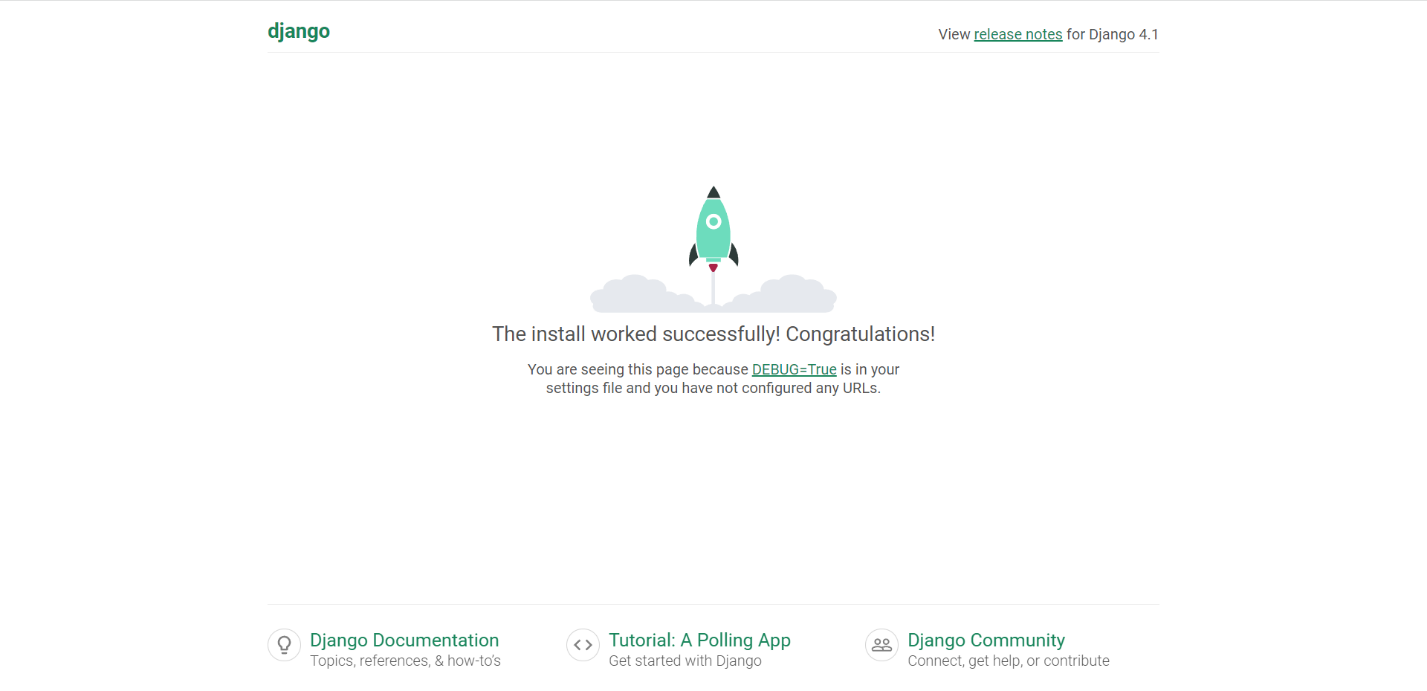
* Примените миграции, используя команду:

poetry run python manage.py migrate

* Запустите сервер, используя команду:

poetry run python manage.py runserver

Приложение будет доступно по адресу [**http://127.0.0.1:8000/**](http://127.0.0.1:8000/).



Теперь вы можете начать разрабатывать ваше Django приложение!

Дефолтная база данных в Django – это SQLite, которая является локальной базой данных, хранящейся в одном файле. В Django этот файл называется db.sqlite3 и он создается автоматически при создании нового проекта.

SQLite обычно используется в маленьких проектах или прототипах, где требуется быстрое создание и прототипирование базы данных без необходимости настройки и установки отдельного сервера баз данных. SQLite может обрабатывать небольшие объемы данных, однако при увеличении нагрузки и объемов данных, она может начать тормозить и не справляться с задачей.

Поэтому, для более серьезной разработки рекомендуется использовать другие СУБД, такие как PostgreSQL, MySQL, Oracle, и т.д. Они обеспечивают более высокую производительность и масштабируемость, а также имеют более широкие возможности для управления и настройки базы данных.

SQLite не следует использовать для production-сайтов, где база данных используется для работы с большим количеством данных и большим количеством пользователей, т.к. это может привести к проблемам производительности и безопасности.

В Django приложения разделяются на отдельные модули, называемые "приложениями" (apps). Каждое приложение отвечает за определенную функциональность веб-сайта, например, аутентификацию пользователей, обработку заказов, управление блогом и т.д.

Приложения в Django могут быть переиспользованы в разных проектах, а также опубликованы в открытом доступе в виде пакетов. Каждое приложение имеет свою собственную структуру каталогов, которая содержит файлы, отвечающие за его функциональность. Структура приложения в Django спроектирована таким образом, чтобы разделить код на логические части, обеспечить его повторное использование и упростить его тестирование.

Стандартная структура каталогов Django-приложения включает следующие файлы и директории:

* Файл models.py: в этом файле определяются модели данных, которые приложение будет использовать для взаимодействия с базой данных.
* Файл views.py: в этом файле определяются функции представлений, которые обрабатывают запросы и возвращают ответы.
* Файлы urls.py: в этом файле определяются маршруты URL, которые соответствуют представлениям.
* Файл admin.py: в этом файле определяются модели данных, которые будут отображаться в административной панели Django.
* Каталог templates: в этом каталоге находятся шаблоны HTML, которые будут использоваться для отображения страниц приложения.
* Каталог static: в этом каталоге находятся статические файлы, такие как CSS, JavaScript, изображения и т.д.
* Файлы apps.py и \_\_init\_\_.py: эти файлы содержат конфигурацию приложения и другие настройки.

Каждое приложение в Django может быть включено в один или несколько проектов. Для этого необходимо определить его в списке INSTALLED\_APPS в файле settings.py проекта.

Когда Django загружает приложение, он выполняет определенные действия, такие как регистрация моделей в базе данных и настройка маршрутов URL-адресов. Кроме того, приложения взаимодействуют друг с другом и с ядром Django через API-интерфейсы, такие как ORM (Object-Relational Mapping) и система шаблонов.

Для создания нового Django приложения необходимо выполнить следующую команду из корневой директории проекта:

poetry run python manage.py startapp <app\_name>

Здесь **<app\_name>** – это имя нового приложения, которое вы хотите создать.

Например, если вы хотите создать приложение с именем "blog", введите следующую команду:

poetry run python manage.py startapp blog

После выполнения этой команды будет создана директория **blog**, содержащая структуру приложения. В ней вы найдете файлы **models.py**, **views.py**, **urls.py**, **admin.py** и др., которые вы можете использовать для создания своего приложения.

Теперь нужно зарегистрировать приложение в файле **settings.py**, добавив его в список **INSTALLED\_APPS**. Для этого откройте файл **settings.py**, найдите переменную **INSTALLED\_APPS** и добавьте в нее строку с именем вашего приложения:

INSTALLED\_APPS = [

# ...

'blog',

# ...

]

После этого Django будет автоматически загружать ваше приложение при запуске проекта.

Теперь вы можете создавать модели в файле **models.py**, создавать представления в файле **views.py**, настраивать маршруты в файле **urls.py** и т.д., используя функциональность Django.

Хорошая структура приложения в Django помогает сохранять код проекта читаемым, легко расширяемым и обеспечивает простоту его тестирования и поддержки в дальнейшем. При инициализации проекта в Django мы создаем структуру проекта, но только одно приложение, которое включает в себя все модели, представления, маршруты и т.д. Но когда проект разрастается, становится сложно управлять всем кодом в одном приложении. Поэтому важно структурировать приложение таким образом, чтобы каждая его часть выполняла свою роль и легко поддерживалась.

Good practice структуры Django-приложения может быть следующей:

1. Создать папку apps для хранения всех ваших приложений Django.
2. Создать папку core для хранения общих модулей, которые будут использоваться в разных приложениях. Например, можете поместить сюда модули, отвечающие за аутентификацию, авторизацию, обработку писем и т.д.
3. Создать отдельные приложения в папке apps. Каждое приложение должно выполнять определенную функцию, например, одно приложение может быть отвечать за пользовательские профили, другое - за управление заказами и т.д. В каждом приложении должны быть модели, представления, маршруты и т.д. связанные только с этим приложением.
4. Разделить настройки проекта для разных стадий приложения (production, development и тд.)

Структурирование приложения таким образом помогает сделать код более организованным и понятным, а также упрощает поддержку проекта в долгосрочной перспективе.

Использование той или иной структуры зависит от потребностей вашего проекта. Например, можно отказаться от части модулей, если того требует задача. Но в рамках данного курса, мы стараемся передать тонкости разработки "боевых проектов". Поэтому и структура будет "боевая":

myproject\_repository/

├── config/

│ ├── .env

│ └── .env.example

├── server/

│ ├── \_\_init\_\_.py

│ ├── apps/

│ ├── core/

│ ├── settings/

│ │ ├── \_\_init\_\_.py

│ │ ├── components/

│ │ │ ├── \_\_init\_\_.py

│ │ │ ├── common.py

│ │ │ └── ...

│ │ └── enviroments/

│ │ ├── \_\_init\_\_.py

│ │ ├── base.py

│ │ ├── local.py.example

│ │ ├── production.py.example

│ │ └── ...

│ ├── urls.py

│ ├── asgi.py

│ └── wsgi.py

├── tests/

├── .gitignore

├── manage.py

├── poetry.lock

├── pyproject.toml

├── readme.md

└── ...

Помимо уже упомянутых в прошлом уроке идеи, тут можно заметить несколько интересных особенностей:

* urls.py, asgi.py, wsgi.py на одном уровне с директориями apps и core
* в корне проекта добавились новые файлы
* в корне проекта добавилась директория config
* settingsтеперь не в виде файла, а в виде отдельного модуля

На первые два пункта постараемся ответить в рамках данного урока, на последние два - в следующем

Приведенная структура проекта наработана опытом, а также готовыми шаблонами, например, wemake-django-template. Чем хороша эта структура?

* Реализация DRY-принципа (Don't Repeat Yourself) и принципа единственной ответственности (Single Responsibility Principle) для уменьшения дублирования кода и повышения модульности проекта.
* Модульность, которая позволяет легко масштабировать и поддерживать проекты

Например, файлы urls.py, asgi.py, wsgi.py на одном уровне с директориями apps и core. А tests на одном уровне с source- основным кодом проекта.

* Все что относится к самому серверу находится на одном уровне. Контекст приложения и его настройки инкапсулированы.
* Сепарация тестов повышает их модульность и изолированность.

Так же в корне проекта добавились новые файлы. Подробно остановимся на каждом из них.

Файл **.gitignore** – это файл, который говорит системе контроля версий Git, какие файлы и папки должны быть проигнорированы при добавлении изменений в репозиторий. Он позволяет исключить ненужные файлы и папки из системы контроля версий и избежать случайного добавления чувствительных данных (например, ключей API, логов или файлов настроек) в репозиторий.

Пример использования **.gitignore** в Django-приложении:

# Ignore all compiled Python files

\*.pyc

# Ignore the virtual environment

venv/

# Ignore enviroment variables

.env

# Ignore the logs directory

logs/

# Ignore the staticfiles directory

staticfiles/

# Ignore the media directory

media/

В этом примере мы исключаем из репозитория все скомпилированные файлы Python (**\*.pyc**), виртуальное окружение (**venv/**), файл с переменными окружения (**.env**),папки с логами (**logs/**), папку со статическими файлами (**staticfiles/**) и папку с медиафайлами (**media/**). Это позволяет не загромождать репозиторий ненужными файлами и защищать конфиденциальные данные.

**settings** в Django – это файлы, которые содержат настройки для приложения. Они определяют параметры, такие как базы данных, пути к файлам, настройки безопасности и другие конфигурационные параметры. Django использует несколько файлов **settings** для управления различными настройками в различных контекстах, например, настройки для разработки, тестирования и продакшн-сервера.

Примеры:

1. **settings.py** – это основной файл настроек Django, который используется для конфигурации основных параметров приложения, таких как базы данных, статические файлы, шаблоны и настройки безопасности.

# secret key - секретный ключ, используемый для подписи куки, токенов и других объектов,

# используемых для безопасности приложения.

SECRET\_KEY = 'qwerty'

# debug - определяет, будет ли включен режим отладки.

# Не следует использовать этот режим в продакшн-сервере.

DEBUG = True

# allowed hosts - список хостов, которые могут обращаться к вашему приложению.

ALLOWED\_HOSTS = ['localhost', '127.0.0.1']

# installed apps - список приложений, которые используются в проекте.

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'myapp',

]

В этом примере мы определяем параметры, такие как секретный ключ, разрешенные хосты, список установленных приложений и отладочный режим. Это базовые параметры, которые нужно определить в **settings.py**.

Best practices:

* Не добавляйте конфиденциальные данные в файлы **settings**, которые необходимо добавлять в систему контроля версий.
* Используйте различные файлы **settings** для разных контекстов, таких как разработка, тестирование и продакшн-сервер.
* Определите базовые параметры в **settings.py** и используйте локальные файлы **settings** для изменения только тех параметров, которые необходимо изменить.
* Никогда не используйте отладочный режим в продакшн-сервере, так как это может представлять угрозу безопасности.
* Не забудьте добавить разрешенные хосты в **ALLOWED\_HOSTS**, чтобы защитить приложение от атаки межсайтовой подделки (CSRF).
* Используйте сильные и случайные значения для секретного ключа, чтобы обеспечить безопасность вашего приложения.\

Мы используем структуру настроек Django, которая предполагает использование модулей Python вместо единственного файла настроек. Это позволяет лучше организовать настройки приложения и разделить их на более мелкие, логически связанные части, облегчает управление настройками приложения в различных контекстах, таких как разработка, тестирование и продакшн-сервер.

Директория **components** содержит модули, которые используются для определения параметров, таких как общие параметры, параметры базы данных, кэш, почта и другие компоненты приложения. В этих модулях определяются параметры конфигурации для каждого компонента.

Директория **environments** содержит файлы **\*.py**, которые определяют параметры для каждого контекста приложения. Например, файл **development.py** определяет настройки для локального разработчика, а файл **production.py** определяет настройки для продакшн-сервера.

Пример:

# settings/components/database.py

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'db\_name',

'USER': 'db\_user',

'PASSWORD': 'db\_password',

'HOST': 'localhost',

'PORT': '5432',

}

}

# settings/environments/development.py

DEBUG = True

ALLOWED\_HOSTS = []

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'db\_name',

'USER': 'db\_user',

'PASSWORD': 'db\_password',

'HOST': 'localhost',

'PORT': '5432',

}

}

В этом примере мы определяем параметры для базы данных в модуле **database.py**. Затем мы определяем настройки для контекста **development** в файле **development.py**, где определяем параметры отладки, разрешенные хосты и базы данных.

Такой подход к настройкам также упрощает сопровождение кода, поскольку разные части настроек хранятся в разных файлах, что позволяет быстро находить и изменять нужные значения, а также добавлять новые настройки без изменения существующего кода.

Переменные окружения – это значения, которые задаются в операционной системе и доступны любой программе, выполняемой в этой среде. В контексте веб-разработки переменные окружения могут использоваться для конфигурирования приложения в различных средах, таких как локальная разработка, тестирование и продакшн.

В Django переменные окружения обычно используются для задания конфиденциальной информации, такой как секретный ключ, пароли для базы данных и других сервисов, а также для настройки параметров, таких как адрес базы данных, порт и другие.

Пример использования переменных окружения в Django:

# settings.py

SECRET\_KEY = os.environ.get('SECRET\_KEY')

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': os.environ.get('DB\_NAME'),

'USER': os.environ.get('DB\_USER'),

'PASSWORD': os.environ.get('DB\_PASSWORD'),

'HOST': os.environ.get('DB\_HOST'),

'PORT': os.environ.get('DB\_PORT', '5432'),

}

}

В этом примере мы используем переменные окружения для задания секретного ключа и параметров базы данных.

Преимущества использования переменных окружения:

* Безопасность: Конфиденциальная информация хранится не в коде, а в окружении, которое может быть настроено для каждого сервера и пользовательской сессии.
* Переносимость: Приложение может быть запущено в разных средах, не требуя изменения кода приложения.
* Гибкость: Переменные окружения могут быть легко изменены без необходимости изменения кода приложения.

Good practices:

* Не храните конфиденциальную информацию в коде приложения. Используйте переменные окружения.
* Используйте уникальные и сложные значения для секретных ключей и других конфиденциальных данных.
* Используйте стандартные имена переменных окружения, такие как **DATABASE\_URL**, **REDIS\_URL**, **SECRET\_KEY** и т.д.
* Храните значения переменных окружения в зашифрованном виде, например, с помощью dotenv или других подобных библиотек.

Файлы .env и .env.example используются для хранения переменных окружения в проекте.

.env файл содержит значения переменных окружения, которые используются в проекте, такие как секретный ключ, настройки базы данных, параметры SMTP и т.д. Этот файл не должен быть добавлен в систему контроля версий, так как он содержит конфиденциальные данные, которые не следует раскрывать для других пользователей или разработчиков.

.env.example файл содержит шаблоны этих переменных окружения, которые должны быть заполнены конкретными значениями перед запуском проекта. Этот файл может быть добавлен в систему контроля версий, чтобы новые разработчики могли легко создать свой собственный файл .env, который соответствует локальной конфигурации.

Использование .env и .env.example файлов позволяет разделить конфиденциальные данные и код, упрощает управление конфигурацией и уменьшает риск случайного коммита конфиденциальных данных в систему контроля версий. Также это позволяет легко переносить проект на другие серверы или на другие окружения, так как конфигурация может быть легко изменена, не затрагивая код.

Помещение файла .env и .env.example в папку config упрощает структуру проекта, позволяет разработчикам быстро найти все настройки конфигурации в одном месте и уменьшает риск случайного коммита конфиденциальных данных в систему контроля версий.

Модели в Django представляют собой классы Python, которые определяют структуру данных для базы данных. Они являются основным способом определения таблиц базы данных и их свойств.

Каждая модель обычно наследуется от базового класса **django.db.models.Model**, что позволяет использовать множество методов, свойств и метаданных для работы с базой данных.

Каждое свойство модели (атрибут класса) представляет столбец таблицы в базе данных, а его тип данных определяется типом поля модели. Например, тип поля **CharField** определяет строковое поле, а тип поля **IntegerField** определяет целочисленное поле.

Пример модели в рамках нашего приложения ведения трат:

from django.db import models

class Operation(models.Model):

name = models.TextField(

verbose\_name="Наименование операции",

)

description = models.TextField(

blank=True,

null=True,

verbose\_name="Описание операции",

)

cost = models.FloatField(

verbose\_name="Стоимость",

)

class Meta:

verbose\_name = "Операция"

verbose\_name\_plural = "Операции"

def \_\_str\_\_(self):

return f"{self.name}"

В этом примере мы создаем модель **Operation**, которая имеет три поля: **name**, **description** и **cost**. Первые два поля представляют строковые и текстовые поля, а третье поле представляет число с плавающей точкой. Также мы определяем метод **\_\_str\_\_()**, который возвращает строковое представление объекта модели.

django.db.models.Model является базовым классом для определения моделей данных в Django. Он представляет таблицу в базе данных, атрибуты которой соответствуют полям таблицы.

В модели также могут быть определены методы, которые позволяют выполнить различные операции с объектами модели, такие как создание, чтение, обновление и удаление (CRUD). Django автоматически создает методы для создания, чтения и обновления объектов модели, а также предоставляет методы для выполнения других операций, таких как удаление, фильтрация, сортировка и т.д.

Важно отметить, что при определении модели, вы можете задавать различные атрибуты, такие как verbose\_name, verbose\_name\_plural, ordering, unique\_together, которые позволяют настроить поведение модели и связи между моделями.

Класс **Meta** является вложенным классом в модель Django, который определяет некоторые метаданные для модели. Он позволяет настраивать поведение модели и ее взаимодействие с базой данных.

Некоторые из основных настроек, которые можно определить в **class Meta**, включают в себя:

* **db\_table**: определяет имя таблицы в базе данных, к которой будет привязана модель.
* **ordering**: определяет порядок, в котором записи модели будут возвращаться из базы данных.
* **verbose\_name** и **verbose\_name\_plural**: определяют человекочитаемые названия модели и ее множественное число.
* **unique\_together**: определяет комбинации полей модели, которые должны быть уникальными в базе данных.

Пример использования **class Meta** для настройки модели:

from django.db import models

class MyModel(models.Model):

# поля модели

class Meta:

db\_table = 'my\_table'

ordering = ['-date\_created']

verbose\_name = 'Моя модель'

verbose\_name\_plural = 'Мои модели'

unique\_together = ('field1', 'field2')

Менеджеры моделей в Django – это классы, которые предоставляют интерфейс для работы с моделями. Они предназначены для управления запросами к базе данных и обработки данных, полученных из базы.

Менеджеры моделей могут быть использованы для:

* Фильтрации объектов модели
* Изменения запросов, возвращающих объекты модели
* Создания и обновления объектов модели
* Выполнения специальных запросов к базе данных

Менеджеры моделей можно использовать как для общих задач, так и для специфических для приложения. В Django есть несколько встроенных менеджеров, но вы также можете создавать свои собственные менеджеры моделей.

Например, создадим модель цели траты:

class GoalType(models.TextChoices):

"""Тип цели"""

SPENDING = "Трата"

REFILL = "Пополнение"

class Goal(models.Model):

"""Модель цели траты или пополнения"""

name = models.CharField(

max\_length=255,

verbose\_name="Название цели",

)

description = models.TextField(

blank=True,

null=True,

verbose\_name="Описание",

)

goal\_type = models.CharField(

choices=enums.GoalType.choices,

default=enums.GoalType.SPENDING,

max\_length=32,

verbose\_name="Тип операций цели",

)

start\_date = models.DateField(

default=date.today,

verbose\_name="Дата начала цели",

)

finish\_date = models.DateField(

blank=True,

null=True,

verbose\_name="Дата окончания цели",

)

value = models.FloatField(

verbose\_name="Значение цели",

)

class Meta:

verbose\_name = "Цель"

verbose\_name\_plural = "Цели"

Чтобы создать менеджер, который возвращает траты и накопления, можно написать следующий класс:

class GoalManager(models.Manager):

def goals(self):

return self.filter(goal\_type=GoalType.REFILL)

def budgets(self):

return self.filter(goal\_type=enums.GoalType.SPENDING)

Далее мы можем использовать этот менеджер в модели,  указав его в свойстве objects:

class Goal(models.Model):

"""Модель цели траты или пополнения"""

name = models.CharField(

max\_length=255,

verbose\_name="Название цели",

)

description = models.TextField(

blank=True,

null=True,

verbose\_name="Описание",

)

goal\_type = models.CharField(

choices=enums.GoalType.choices,

default=enums.GoalType.SPENDING,

max\_length=32,

verbose\_name="Тип операций цели",

)

start\_date = models.DateField(

default=date.today,

verbose\_name="Дата начала цели",

)

finish\_date = models.DateField(

blank=True,

null=True,

verbose\_name="Дата окончания цели",

)

value = models.FloatField(

verbose\_name="Значение цели",

)

objects = GoalManager()

class Meta:

verbose\_name = "Цель"

verbose\_name\_plural = "Цели"

Теперь мы можем использовать кастомные методы следующим образом:

goals = Goal.objects.goals()

budgets = Goal.objects.budgets()

В результате мы получим отдельно цели накопления и цели трат пользователя.

Миграции в Django – это способ управления изменениями в структуре базы данных приложения. Django создает миграции автоматически при внесении изменений в модели. Миграции можно применять к базе данных для того, чтобы изменения структуры моделей были отражены в базе данных.

Пример:

Предположим, что у нас есть модель **Operation**с полями **name**, **description**, **cost**. После того, как мы начали использовать эту модель, мы решили добавить в нее новое поле **operation\_at**. Мы можем сделать это, изменив определение модели:

from django.db import models

from datetime import datetime

class Operation(models.Model):

name = models.TextField(

verbose\_name="Наименование операции",

)

description = models.TextField(

blank=True,

null=True,

verbose\_name="Описание операции",

)

cost = models.FloatField(

verbose\_name="Стоимость",

)

operation\_at = models.DateTimeField(

default=datetime.now,

verbose\_name="Дата операции",

)

class Meta:

verbose\_name = "Операция"

verbose\_name\_plural = "Операции"

def \_\_str\_\_(self):

return f"At-{self.operation\_at}-{self.name}"

Теперь нам нужно сказать Django, как изменить структуру базы данных для отображения нового поля. Для этого мы можем создать миграцию, которая добавит новое поле **operation\_at** в таблицу базы данных, соответствующую модели **Operation**. Мы можем создать эту миграцию следующей командой:

poetry run python manage.py makemigrations

После выполнения этой команды будет создана миграция:

import datetime

from django.db import migrations, models

class Migration(migrations.Migration):

dependencies = [

("operations", "0001\_initial"),

]

operations = [

migrations.AddField(

model\_name='operation',

name='operation\_at',

field=models.DateTimeField(

default=datetime.datetime.now,

verbose\_name="Дата операции",

),

),

]

Её можно применить к базе данных следующей командой:

poetry run python manage.py migrate

Эта команда применит все недостающие миграции, включая новую миграцию для добавления поля **operation\_at** .

Миграции также позволяют откатывать изменения в структуре базы данных, если это необходимо. Для этого мы можем использовать команду **poetry run python manage.py migrate <app\_name> <migration\_name>** с указанием имени миграции, к которой нужно откатиться.

Миграции позволяют удобно управлять структурой базы данных в проекте Django и вносить изменения без необходимости изменения структуры базы данных вручную.

Помимо автоматических миграций данных, иногда миграции создают самостоятельно. Чаще всего это нужно для миграции данных между таблицами\моделями. Для этого нужно:

* Создать миграцию, используя команду **poetry run python manage.py makemigrations <app\_name> --empty <migration\_name>**:
* Открыть созданный файл миграции.
* Добавить необходимые изменения в функцию **operations**.

from django.db import migrations

def add\_data(apps, schema\_editor):

...

class Migration(migrations.Migration):

dependencies = [

('myapp', '0001\_initial'),

]

operations = [

migrations.RunPython(add\_data),

* Применить миграцию с помощью команды

Теперь мы можем проверить, что новые записи добавлены в базу данных.

Админ-панель – это встроенный в Django интерфейс для управления данными моделей в приложении. Она позволяет администраторам управлять контентом, изменять настройки приложения и выполнять другие задачи без необходимости взаимодействия с базой данных напрямую.

Для использования админ-панели необходимо зарегистрировать модели в файле **admin.py**, который обычно находится в приложени. Пример файла **admin.py** для модели **MyModel**:

from django.contrib import admin

from .models import MyModel

@admin.register(MyModel)

class MyModelAdmin(admin.ModelAdmin):

list\_display = ('id', 'name', 'created\_at') # поля для отображения в списке объектов

list\_filter = ('created\_at',) # фильтр по дате создания

search\_fields = ('name',) # поиск по имени

ordering = ('-created\_at',) # сортировка по дате создания

Здесь мы импортируем модель **MyModel** из модуля **models**, регистрируем ее с помощью декоратора **admin.register** и создаем класс **MyModelAdmin**, который наследуется от **admin.ModelAdmin**. В этом классе мы определяем, какие поля модели должны отображаться в списке объектов (**list\_display**), какие фильтры и поиск следует использовать (**list\_filter** и **search\_fields**) и как сортировать список объектов (**ordering**).

Не обязательно создавать собственный класс **MyModelAdmin.** Можно воспользоваться просто командой **admin.site.register(MyModel)**

Тогда модель добавиться автоматически в admin панель, без необходимости описания дополнительных параметров.

Некоторые методы, которые можно переопределить, чтобы изменить стандартное поведение интерфейса.

1. get\_queryset(self, request) – возвращает QuerySet, который будет использоваться для отображения объектов модели в административной панели. Можно переопределить этот метод для изменения списка объектов, которые будут отображаться в административной панели.
2. get\_fields(self, request, obj=None) – возвращает список полей, которые будут отображаться для каждого объекта модели. Можно переопределить этот метод, чтобы изменить список полей для каждого объекта.
3. get\_readonly\_fields(self, request, obj=None) – возвращает список полей, которые будут только для чтения. Можно переопределить этот метод, чтобы изменить список полей только для чтения.
4. get\_form(self, request, obj=None, \*\*kwargs) – возвращает форму, которая будет использоваться для создания или изменения объектов модели. Можно переопределить этот метод, чтобы изменить форму или добавить свои собственные поля.
5. save\_model(self, request, obj, form, change) – вызывается после того, как объект модели был сохранен. Можно переопределить этот метод, чтобы добавить свою логику сохранения.
6. delete\_model(self, request, obj) – вызывается перед удалением объекта модели. Можно переопределить этот метод, чтобы добавить свою логику удаления.
7. get\_actions(self, request) – возвращает список доступных действий над объектами модели. Можно переопределить этот метод, чтобы изменить список доступных действий.
8. get\_urls(self) – возвращает список URL-адресов, которые будут использоваться для отображения объектов модели в административной панели. Можно переопределить этот метод, чтобы добавить свои собственные URL-адреса.
9. has\_change\_permission(self, request, obj=None) – возвращает значение True, если пользователь имеет право изменять объекты модели. Можно переопределить этот метод, чтобы изменить права доступа.
10. has\_delete\_permission(self, request, obj=None) – возвращает значение True, если пользователь имеет право удалять объекты модели. Можно переопределить этот метод, чтобы изменить права доступа.
11. has\_add\_permission(self, request) – возвращает значение True, если пользователь имеет право создавать объекты модели. Можно переопределить этот метод, чтобы изменить права доступа.

Некоторые атрибуты, которые можно переопределить, чтобы изменить стандартное поведение интерфейса.

1. list\_display – определяет список полей, которые будут отображаться в списке объектов модели. Этот атрибут может содержать названия полей модели или имена методов класса ModelAdmin, которые будут вызываться для отображения дополнительной информации.
2. list\_filter – определяет список полей, которые будут использоваться для фильтрации объектов модели. Этот атрибут может содержать названия полей модели или имена методов класса ModelAdmin, которые будут вызываться для фильтрации дополнительной информации.
3. search\_fields – определяет список полей, которые будут использоваться для поиска объектов модели. Этот атрибут может содержать названия полей модели или имена методов класса ModelAdmin, которые будут вызываться для поиска дополнительной информации.
4. ordering – определяет порядок сортировки объектов модели по умолчанию. Этот атрибут может содержать названия полей модели или имена методов класса ModelAdmin, которые будут вызываться для сортировки объектов.
5. list\_select\_related – определяет связанные объекты модели, которые будут загружены одновременно с основными объектами. Этот атрибут может содержать названия связанных полей модели.
6. list\_per\_page – определяет количество объектов модели, которые будут отображаться на странице.
7. list\_max\_show\_all – определяет максимальное количество объектов модели, которые могут быть отображены на одной странице, если пользователь выберет опцию "Show all".
8. list\_editable – определяет список полей модели, которые можно редактировать в списке объектов модели.
9. actions – определяет список доступных действий над объектами модели. Этот атрибут должен быть итерируемым объектом, состоящим из методов класса ModelAdmin.
10. actions\_on\_top – определяет, будут ли доступны действия над объектами модели на верхней части страницы. Значение по умолчанию – True.
11. actions\_on\_bottom – определяет, будут ли доступны действия над объектами модели на нижней части страницы. Значение по умолчанию – False.