

Оглавление

КРИТЕРИИ.....	2	Задание.....	42	ЗАДАНИЕ.....	79
ПРАКТИЧЕСКОЕ ЗАДАНИЕ		Контрольные вопросы.....	42	Контрольные вопросы.....	79
№1.....	3	ПРАКТИЧЕСКОЕ ЗАДАНИЕ		ПРАКТИЧЕСКОЕ ЗАДАНИЕ	
Концепция Django Framework	3	№6.....	43	№11.....	80
Теоретическая часть	3	DjangoOrm запросы.....	43	Формы django и регистрация	80
Практическая часть	10	Теоретическая часть	43	ПРАКТИЧЕСКАЯ ЧАСТЬ	80
Задание.....	13	ПРАКТИЧЕСКАЯ ЧАСТЬ	48	ЗАДАНИЕ.....	84
Контрольные вопросы.....	13	Задание.....	51	Контрольные вопросы.....	84
ПРАКТИЧЕСКОЕ ЗАДАНИЕ		ПРАКТИЧЕСКОЕ ЗАДАНИЕ		ПРАКТИЧЕСКОЕ ЗАДАНИЕ	
№2.....	14	№7.....	52	№12.....	85
Маршрутизация и рендер html шаблонов.....	14	Шаблонизатор в Django, статические файлы.....	52	Авторизация.....	85
Теоретическая часть	14	Теоретическая часть	52	ПРАКТИЧЕСКАЯ ЧАСТЬ	85
Практическая часть	18	ПРАКТИЧЕСКАЯ ЧАСТЬ	55	ЗАДАНИЕ.....	87
Задание.....	22	Задание.....	59	Контрольные вопросы.....	87
Контрольные вопросы.....	22	ПРАКТИЧЕСКОЕ ЗАДАНИЕ		ПРАКТИЧЕСКОЕ ЗАДАНИЕ	
ПРАКТИЧЕСКОЕ ЗАДАНИЕ		№8.....	60	№13.....	88
№3.....	23	Ссылки и медиафайл.....	60	Аутентификации.....	88
Создание моделей.....	23	Теоретическая часть	60	Теоретическая часть	88
Теоретическая часть	23	ПРАКТИЧЕСКАЯ ЧАСТЬ	64	ПРАКТИЧЕСКАЯ ЧАСТЬ	89
Практическая часть	27	ЗАДАНИЕ.....	67	ЗАДАНИЕ.....	93
Задание.....	30	Контрольные вопросы.....	67	Контрольные вопросы.....	93
Контрольные вопросы.....	30	ПРАКТИЧЕСКОЕ ЗАДАНИЕ		ПРАКТИЧЕСКОЕ ЗАДАНИЕ	
ПРАКТИЧЕСКОЕ ЗАДАНИЕ		№9.....	68	№14.....	94
№4.....	31	Динамические ссылки.....	68	Кастомные сессии и context_processors.....	94
Связи.....	31	Теоретическая часть	68	Теоретическая часть	94
Теоретическая часть	31	ПРАКТИЧЕСКАЯ ЧАСТЬ	71	ПРАКТИЧЕСКАЯ ЧАСТЬ	98
Практическая часть	33	ЗАДАНИЕ.....	73	ЗАДАНИЕ.....	104
Задание.....	35	Контрольные вопросы.....	73	Контрольные вопросы.....	104
Контрольные вопросы.....	35	ПРАКТИЧЕСКОЕ ЗАДАНИЕ		ПРАКТИЧЕСКОЕ ЗАДАНИЕ	
ПРАКТИЧЕСКОЕ ЗАДАНИЕ		№10.....	74	№15.....	105
№5.....	36	Http запросы.....	74	TabularInline админ и создание заказов.....	105
Админ-панель.....	36	Теоретическая часть	74	ПРАКТИЧЕСКАЯ ЧАСТЬ	105
Теоретическая часть	36	ПРАКТИЧЕСКАЯ ЧАСТЬ	78		

КРИТЕРИИ

Выполненные действия	Оценка
Практическая часть или Теоретическая часть	3
Практическая часть или Теоретическая часть + Контрольные вопросы	4
Практическая часть + Теоретическая часть + Контрольные вопросы	5

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №1

Концепция Django Framework

ЦЕЛЬ ЗАНЯТИЯ

Изучить концепцию django и веб-приложение «Hello, World»

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Что такое Django?

Django — это высокоуровневый Python веб-фреймворк, который позволяет быстро создавать безопасные и поддерживаемые веб-сайты. Созданный опытными разработчиками, Django берёт на себя большую часть хлопот веб-разработки, поэтому вы можете сосредоточиться на написании своего веб-приложения без необходимости изобретать велосипед. Он бесплатный и с открытым исходным кодом, имеет растущее и активное сообщество, отличную документацию и множество вариантов как бесплатной, так и платной поддержки.

Django помогает писать программное обеспечение, которое будет:

Полным

Django следует философии «Всё включено» и предоставляет почти всё, что разработчики могут захотеть сделать «из коробки». Поскольку всё, что вам нужно, является частью единого «продукта», всё это безупречно работает вместе, соответствует последовательным принципам проектирования и имеет обширную и актуальную документацию.

Разносторонним

Django может быть (и был) использован для создания практически любого типа веб-сайтов — от систем управления контентом и wiki до социальных сетей и новостных сайтов. Он может работать с любой клиентской средой и может доставлять контент практически в любом формате (включая HTML, RSS-каналы, JSON, XML и т. д.). Сайт, который вы сейчас читаете, создан с помощью Django!

Хотя Django предоставляет решения практически для любой функциональности, которая вам может понадобиться (например, для нескольких популярных баз данных, шаблонизаторов и т. д.), внутренне он также может быть расширен сторонними компонентами, если это необходимо.

Безопасным

Django помогает разработчикам избежать многих распространённых ошибок безопасности, предоставляя фреймворк, разработанный чтобы «делать правильные вещи» для автоматической защиты сайта. Например, Django предоставляет безопасный способ управления учётными записями пользователей и паролями, избегая распространённых ошибок, таких как размещение информации о сеансе в файлы cookie, где она уязвима (вместо этого файлы cookie содержат только ключ, а фактические данные хранятся в базе данных) или непосредственное хранение паролей вместо хэша пароля.

Django, по умолчанию, обеспечивает защиту от многих уязвимостей, включая SQL-инъекцию, межсайтовый скрипting, подделку межсайтовых запросов и кликджекинг (см. Website security для получения дополнительной информации об этих атаках).

Масштабируемым

Django использует компонентную "shared-nothing" архитектуру (каждая её часть независима от других и, следовательно, может быть заменена или изменена, если это необходимо). Чёткое разделение частей означает, что Django может масштабироваться при увеличении трафика, путём добавления оборудования на любом уровне: серверы кеширования, серверы баз данных или серверы приложений. Одни из самых загруженных сайтов успешно масштабировали Django (например, Instagram и Disqus, если назвать только два из них).

Удобным в сопровождении

Код Django написан с использованием принципов и шаблонов проектирования, которые поощряют создание поддерживаемого и повторно используемого кода. В частности, в нём используется принцип «Don't Repeat Yourself» (DRY, «не повторяйся»), поэтому нет ненужного дублирования, что сокращает объём кода. Django также способствует группированию связанных функциональных возможностей в повторно используемые «приложения» и, на более низком уровне, группирует связанный код в модули.

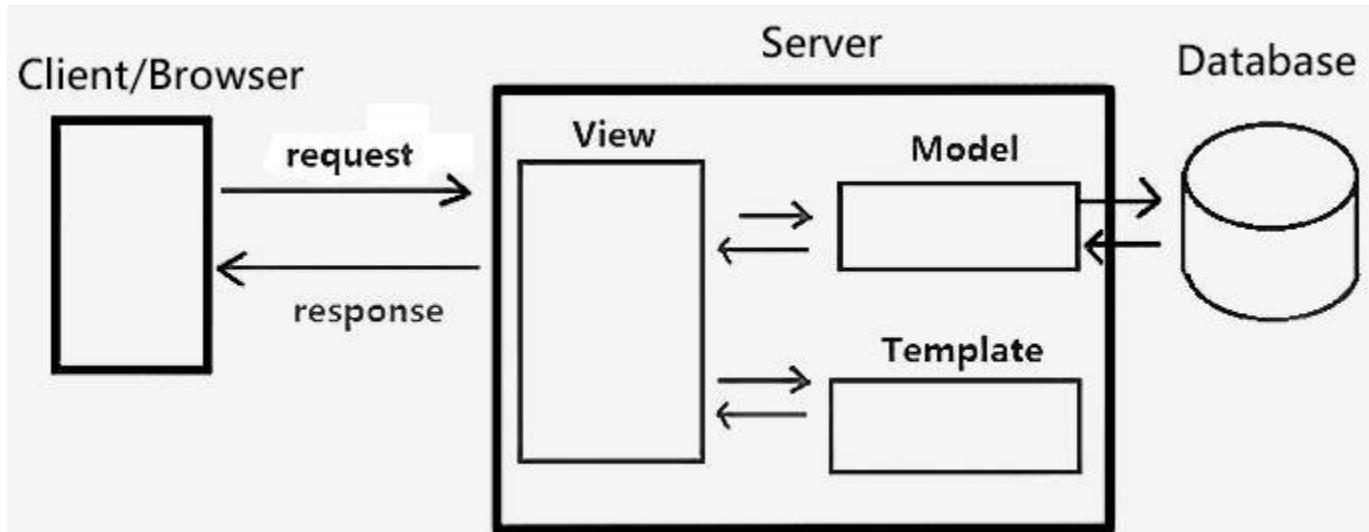


Рисунок 1 - Шаблон проектирования MTV

Переносным

Django написан на Python, который работает на многих платформах. Это означает, что вы не привязаны к какой-либо конкретной серверной платформе и можете запускать приложения на многих версиях Linux, Windows и Mac OS X. Кроме того, Django хорошо поддерживается многими веб-хостингами, которые часто предоставляют определённую инфраструктуру и документацию для размещения сайтов Django.

Как выглядит код Django?

На традиционном информационном веб-сайте веб-приложение ожидает HTTP-запросы от веб-браузера (или другого клиента). Когда запрос получен, приложение разрабатывает то, что необходимо на основе URL-адреса и, возможно, данных в POST или GET запросах. В зависимости от того, что требуется, далее он может читать или записывать информацию из базы данных или выполнять другие задачи, необходимые для удовлетворения запроса. Затем приложение вернёт ответ веб-браузеру, часто динамически создавая HTML-страницу для отображения в браузере, вставляя полученные данные в HTML-шаблон.

Веб-приложения, написанные на Django, обычно группируют код, который обрабатывает каждый из этих шагов, в отдельные файлы:

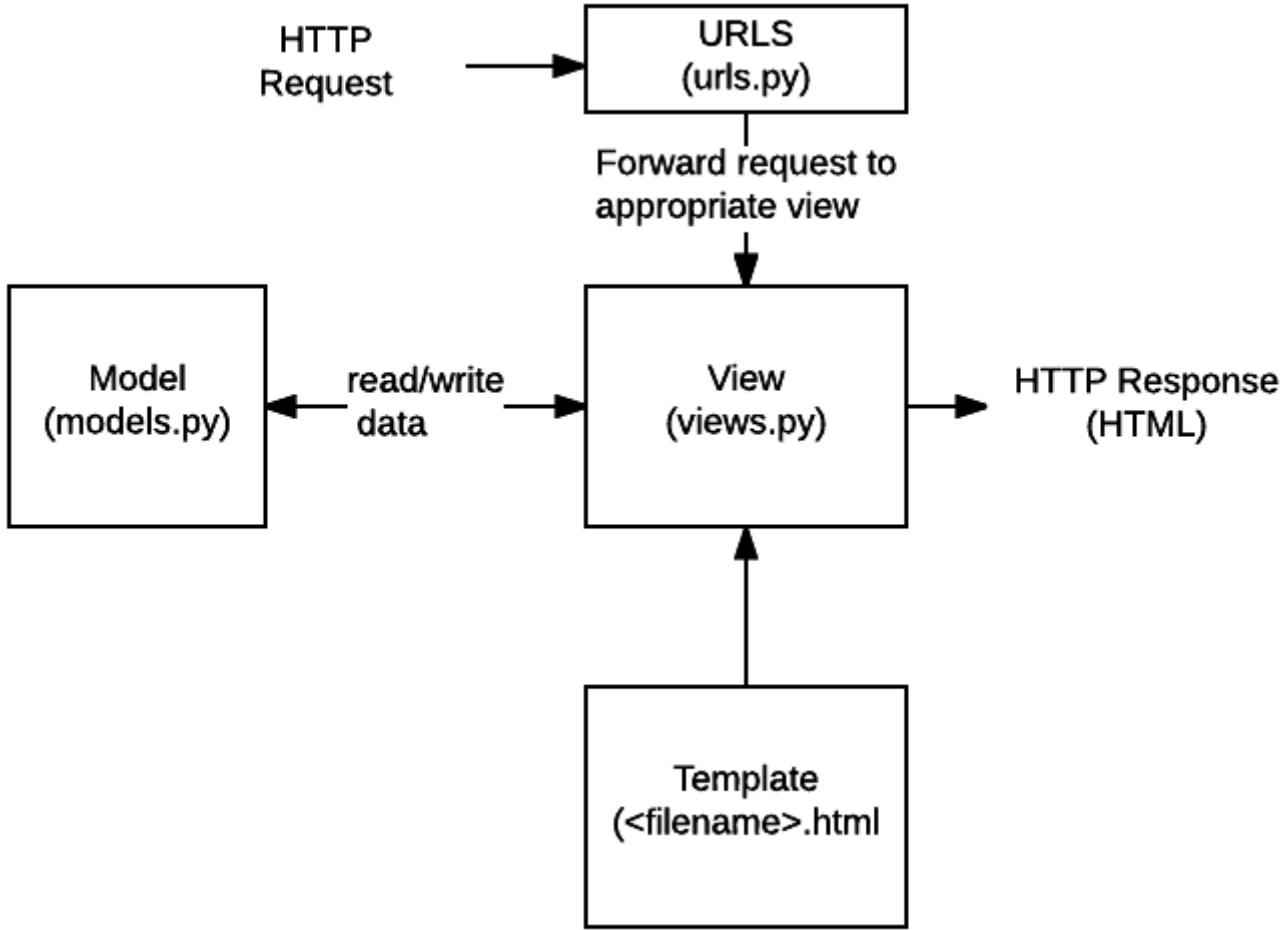


Рисунок 2: Архитектура приложения Django

- **URLs:** Хотя можно обрабатывать запросы с каждого URL-адреса с помощью одной функции, гораздо удобнее писать отдельную функцию для обработки каждого ресурса. URL-маршрутизатор используется для перенаправления HTTP-запросов в соответствующее представление на основе URL-адреса запроса. Кроме того, URL-маршрутизатор может извлекать данные из URL-адреса в соответствии с заданным шаблоном и передавать их в соответствующую функцию отображения (view) в виде аргументов.
- **View:** View (англ. «отображение») — это функция обработчика запросов, которая получает HTTP-запросы и возвращает ответы. Функция view имеет доступ к данным, необходимым для удовлетворения запросов, и делегирует ответы в шаблоны через модели.
- **Models:** Модели представляют собой объекты Python, которые определяют структуру данных приложения и предоставляют механизмы для управления (добавления, изменения, удаления) и выполнения запросов в базу данных.
- **Templates:** Template (англ. «шаблон») — это текстовый файл, определяющий структуру или разметку страницы (например HTML-страницы), с полями для подстановки, которые используются для вывода актуального содержимого. View может динамически создавать HTML-страницы, используя HTML-шаблоны и заполняя их данными из модели (model). Шаблон может быть использован для определения структуры файлов любых типов, не обязательно HTML.

Следующие разделы дадут вам понимание того, как выглядят основные части Django (мы их изучим более детально чуть позже на курсе, когда будет настраивать окружение разработчика).

Отправка запроса в правильное view (urls.py)

Сопоставитель URL-адресов обычно содержится в файле **urls.py**. В примере ниже сопоставитель (**urlpatterns**) определяет список сопоставлений между **маршрутами** (определенными URL-шаблонами) и

соответствующими функциями отображения (view). Если получен HTTP-запрос, который имеет URL-адрес, соответствующий определённому шаблону, то затем будет вызвана связанная функция отображения (view) и передана в запрос.

```
1 urlpatterns = [
2     path('admin/', admin.site.urls),
3     path('book/<int:id>', views.book_detail, name='book_detail'),
4     path('catalog/', include('catalog.urls')),
5     re_path(r'^([0-9]+)/$', views.best),
6 ]
```

Рисунок 3 - Файл urls.py

Объект urlpatterns является списком функций path() и/или re_path() (в Python списки определяются с помощью квадратных скобок, внутри которых элементы разделены запятыми и могут содержать необязательную завершающую запятую. Например: [item1, item2, item3,]).

Первый аргумент в обоих методах - маршрут (шаблон), который будет сопоставлен. В методе path() угловые скобки используются для определения частей URL-адреса, которые будут захвачены и переданы в функцию отображения (view) в качестве именованных аргументов. Функция re_path() использует гибкий подход к сопоставлению с шаблоном, известный как регулярное выражение. Мы поговорим об этом в следующей статье!

Второй аргумент — это ещё одна функция, которая будет вызываться при сопоставлении шаблона. Обозначение views.book_detail указывает, что функция называется book_detail() и может быть обнаружена в модуле с именем views (т.е. внутри файла с именем views.py).

Обработка запроса (views.py)

Отображения (views) — это сердце веб-приложения, принимающего HTTP-запросы от веб-клиентов и возвращающего HTTP-ответы. Между этим они используют другие ресурсы фреймворка для доступа к базам данных, шаблонам визуализации и т. д.

В приведённом ниже примере показана минимальная функция представления index(), которая могла быть вызвана нашим сопоставителем URL-адресов в предыдущем разделе. Как и все функции отображения (view), она получает объект HttpRequest в качестве параметра (request) и возвращает объект HttpResponse. В этом случае мы ничего не делаем с запросом, и наш ответ просто возвращает жёстко запрограммированную строку. Мы покажем вам запрос, который делает что-то более интересное в следующем разделе.



```
1 from django.http import HttpResponseRedirect  
2  
3 def index(request):  
4     # Получить HttpRequest – параметр запроса  
5     # Выполнить операции, используя информацию из запроса.  
6     # Вернуть HttpResponseRedirect  
7     return HttpResponseRedirect('Hello from Django!')
```

Рисунок 4 - Файл views.py

Определение данных модели (models.py)

Веб-приложения Django обрабатывают и запрашивают данные через объекты Python, называемые моделями. Модели определяют структуру хранимых данных, включая типы полей и, возможно, их максимальный размер, значения по умолчанию, параметры списка выбора, текст справки для документации, текст меток для форм и т. д. Определение модели не зависит от используемой базы данных — ваши модели будут работать в любой из них. После того как вы выбрали базу данных, которую хотите использовать, вам не нужно напрямую обращаться к ней — вы просто пишете свою структуру модели и другой код, а Django выполняет всю «грязную работу» по обращению к базе данных за вас.

В приведённом ниже фрагменте кода показана очень простая модель Django для объекта Team. Класс Team наследуется от класса models.Model. Он определяет имя команды и командный уровень в качестве полей символов и задаёт максимальное количество символов, которые могут быть сохранены для каждой записи. Team_level может быть одним из нескольких значений, поэтому мы определяем его как поле выбора и предоставляем сопоставление между отображаемыми вариантами и хранимыми данными вместе со значением по умолчанию.



```
1  from django.db import models
2
3  class Team(models.Model):
4      team_name = models.CharField(max_length=40)
5
6      TEAM_LEVELS = (
7          ('U09', 'Under 09s'),
8          ('U10', 'Under 10s'),
9          ('U11', 'Under 11s'),
10         ... #список других командных уровней
11     )
12     team_level = models.CharField(max_length=3, choices=TEAM_LEVELS, default='U11')
13
```

Рисунок 5 - Файл *models.py*

Запросы данных (views.py)

Модель Django предоставляет простой API запросов для поиска в базе данных. Поиск может осуществляться по нескольким полям одновременно, используя различные критерии (такие как exact («точный»), case-insensitive («без учёта регистра»), greater than («больше чем») и т. д.), и может поддерживать сложные выражения (например, вы можете указать поиск в командах U11, у которых есть имя команды, начинающееся с «Fr» или заканчивающееся на «al»).

Фрагмент кода показывает функцию view (обработчик ресурсов) для отображения всех команд U09. Выделенная жирным строка показывает, как мы можем использовать модель API-запросов для того, чтобы отфильтровать все записи, где поле `team_level` в точности содержит текст 'U09' (обратите внимание, как эти критерии передаются функции `filter()` в качестве аргумента с именем поля и типом соответствия, разделённым двойным подчёркиванием: `team_level__exact`).



```
1 from django.shortcuts import render
2 from .models import Team
3
4 def index(request):
5     list_teams = Team.objects.filter(team_level_exact="U09")
6     context = {'youngest_teams': list_teams}
7     return render(request, '/best/index.html', context)
8
```

Рисунок 6 - Измененый файл views.py

Данная функция использует функцию render() для того, чтобы создать HttpResponse, который будет отправлен назад браузеру. Эта функция является ярлыком; она создаёт HTML-файл, комбинируя указанный HTML-шаблон и некоторые данные для вставки в шаблон (предоставляется в переменной с именем «context»). В следующем разделе мы покажем как данные вставляются в шаблон для создания HTML-кода.

Вывод данных (HTML-шаблоны)

Системы шаблонов позволяют указать структуру выходного документа, используя заполнители для данных, которые будут вставлены при генерировании страницы. Шаблоны часто используются для создания HTML, но также могут создавать другие типы документов. Django «из коробки» поддерживает как собственную систему шаблонов, так и другую популярную библиотеку Python под названием Jinja2 (она также может быть использована для поддержки других систем, если это необходимо).

Фрагмент кода показывает, как может выглядеть HTML-шаблон, вызванный функцией render() из предыдущего раздела. Этот шаблон был написан с предположением, что во время отрисовки он будет иметь доступ к переменной списка, названной youngest_teams (содержащейся в контекстной переменной внутри функции render() выше). Внутри скелета HTML мы имеем выражение, которое сначала проверяет, существует ли переменная youngest_teams, а затем повторяет её в цикле for. При каждом повторе шаблон отображает значение team_name каждой команды в элементе .



```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <title>Home page</title>
6  </head>
7  <body>
8      {% if youngest_teams %}
9          <ul>
10             {% for team in youngest_teams %}
11                 <li>{{ team.team_name }}</li>
12             {% endfor %}
13         </ul>
14     {% else %}
15         <p>No teams are available.</p>
16     {% endif %}
17  </body>
18  </html>
```

Рисунок 7 - Файл index.html

ПРАКТИЧЕСКАЯ ЧАСТЬ

В этой практической мы создадим простое веб-приложение «Hello, World». Для этого создаем виртуальное окружение, активируем виртуальное окружение, и скачиваем Django.

1. Пишем в терминале команду:

django-admin startproject название_проекта(например app)

2. Заходим в директорию(папка) app с помощью команды:

```
cd название_директории
```

3. Запускаем сервер с помощью команды:

```
py manage.py runserver
```

После ввода команды у вас запустить локальный сервер и выводится сообщение.

```
Django version 5.0.4, using settings 'app.settings'  
Starting development server at http://127.0.0.1:8000/  
Quit the server with CTRL-BREAK.
```

Рисунок 8 - Сообщение о запуске локального сервера

Чтобы открыть веб-приложение, зажмите CTRL и нажмите ЛКМ на <http://127.0.0.1:8000/> в терминале. После в этих действий у вас откроется браузер с веб-страницей:

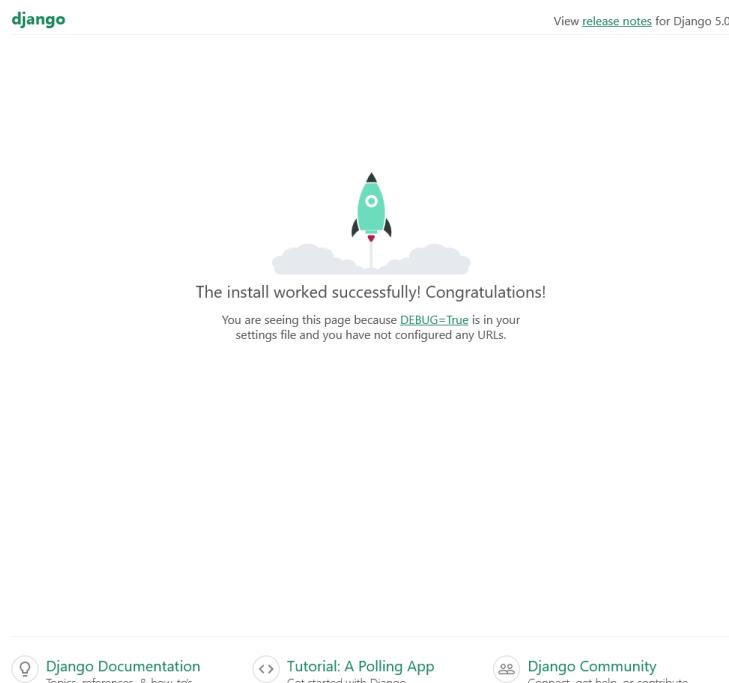


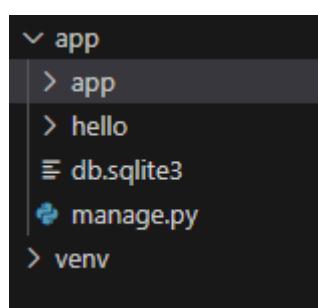
Рисунок 9 - Стандартная веб-страница django

4. После этих действий создаем приложение. (Чтобы выключить сервер нажмите CTRL+C)

Для этого пишем в терминале:

```
py manage.py startapp название_приложения(например hello)
```

После этих действий в папке проекта появится директория с названием hello



*Рисунок 10 -
Структура проекта*

5. Дальше заходим в файл views.py, которая находится в папке hello и пишем данный код.

```
1 from django.shortcuts import HttpResponse
2
3 # Create your views here.
4
5
6 def get_hello(request):
7     return HttpResponse('<h1>Hello world!</h1>')
```

Рисунок 11 - Функция для отрисовки страницы

6. В папке app, заходим в файл urls.py. В файле urls.py создаем ссылки для веб-приложения, с помощью функции path. В функцию path передаем два аргумента:

1. Название ссылки, например в данном примере пустая строчка, и это означает что первая страница нашего сайта
2. Функция, которая передает данные(В данном примере у нас html тэг и функция get_hello, которую мы импортируем из файла views.py)

```
1 from django.contrib import admin
2 from django.urls import path
3 from hello.views import get_hello
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6     path('', get_hello)
7 ]
```

Рисунок 12 - Файл urls.py

7. Запускаем сервер

Hello world!

Рисунок 13 - Веб-приложение "Hello, World!"

Задание

1. Создайте приложение privet
2. Создайте веб-страницу по ссылкой `privet(http://127.0.0.1:8000/privet)`, и по переходе на эту будет показываться «Привет»

Контрольные вопросы

1. В каком файле обрабатываются http запросы?
2. В каком файле отвечает за маршрутизацию?
3. С помощью какой команды создается проект на Django?
4. С помощью какой команды создается приложение для проекта на Django?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №2

Маршрутизация и рендер html шаблонов

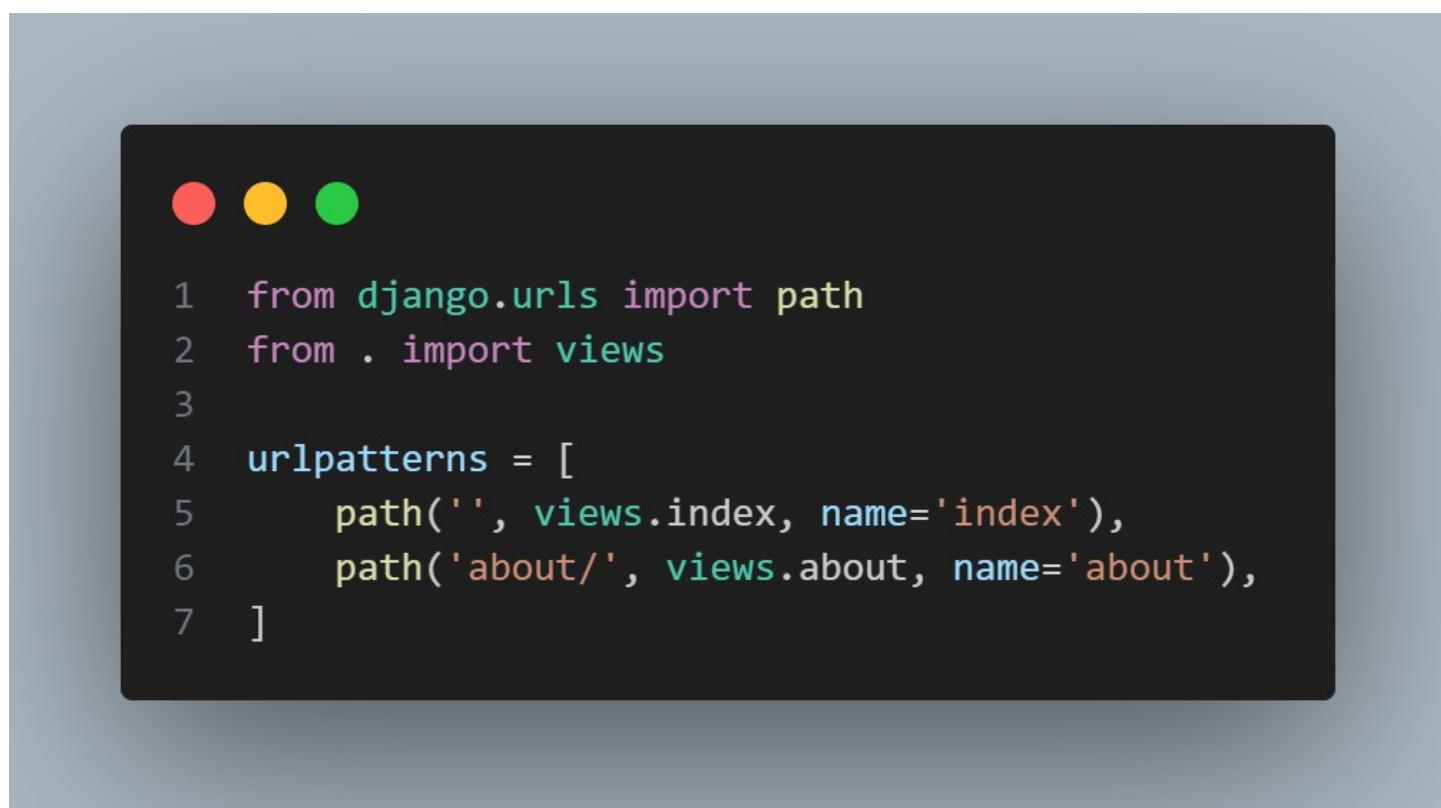
ЦЕЛЬ ЗАНЯТИЯ

1. Изменить метод использования маршрутизации для файла urls.py
2. Познакомиться с функцией render

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

В Django маршрутизация играет ключевую роль в сопоставлении URL-запросов с соответствующими обработчиками (views). Файл urls.py отвечает за определение URL-шаблонов и привязку их к соответствующим обработчикам. В данной теоретической части рассмотрим, как изменить и настроить маршрутизацию в файле urls.py.

Основы маршрутизации



```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('', views.index, name='index'),
6     path('about/', views.about, name='about'),
7 ]
```

Рисунок 14 - Файл urls.py

В типичном проекте Django файл urls.py выглядит примерно так:

Изменение метода маршрутизации

Для более гибкой маршрутизации и улучшения структуры проекта можно использовать следующие методы:

1. **Использование функции include:** Для больших проектов целесообразно разбивать маршруты на логические группы и использовать функцию include для подключения маршрутов из других файлов urls.py.



```
1 from django.urls import path, include
2
3 urlpatterns = [
4     path('blog/', include('blog.urls')),
5     path('shop/', include('shop.urls')),
6 ]
7
```

Рисунок 15 - Основной файл urls.py

В файлах blog/urls.py и shop/urls.py определяются маршруты для соответствующих приложений:



```
1 # blog/urls.py
2 from django.urls import path
3 from . import views
4
5 urlpatterns = [
6     path('', views.blog_index, name='blog_index'),
7     path('<int:post_id>/', views.blog_detail, name='blog_detail'),
8 ]
9
```

Рисунок 16 - Файл urls.py в приложении blog

2. Использование регулярных выражений: Django поддерживает использование регулярных выражений для более сложных маршрутов.



```
1 from django.urls import re_path
2 from . import views
3
4 urlpatterns = [
5     re_path(r'^archive/(?P<year>[0-9]{4})/$', views.archive, name='archive'),
6 ]
7
```

Рисунок 17 - Использование функции `re_path`

3. **Именованные группы:** Именованные группы в регулярных выражениях позволяют передавать параметры в обработчики.



```
1 from django.urls import re_path
2 from . import views
3
4 urlpatterns = [
5     re_path(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.article_month, name='article_month'),
6 ]
7
```

Рисунок 18 - Использование функции `re_path` с регулярными выражениями

Преимущества изменения метода маршрутизации

- **Модульность:** Разделение маршрутов по приложениям делает проект более организованным и поддерживаемым.
- **Гибкость:** Использование регулярных выражений позволяет обрабатывать более сложные URL-шаблоны.
- **Переиспользование кода:** Функция `include` позволяет легко подключать и переиспользовать маршруты.

2. Познакомиться с функцией `render`

Функция `render` является одной из ключевых функций в Django, используемых для рендеринга HTML-шаблонов и формирования ответов на запросы. Она упрощает процесс создания ответа, объединяя данные контекста с HTML-шаблоном и возвращая готовый HTTP-ответ.

Основы использования функции `render`

Функция `render` импортируется из модуля `django.shortcuts` и используется следующим образом:



```
1 from django.shortcuts import render
2
3 def index(request):
4     context = {'title': 'Главная страница', 'content': 'Добро пожаловать на наш сайт!'}
5     return render(request, 'index.html', context)
```

Рисунок 19 - Использование функции render

Параметры функции render

1. **request:** Объект запроса, передаваемый во view.
2. **template_name:** Имя HTML-шаблона, который будет рендериться.
3. **context (необязательный):** Словарь с данными, которые будут переданы в шаблон.

Преимущества использования render

- **Простота:** Функция render объединяет несколько шагов в один, делая код более читабельным и упрощая процесс разработки.
- **Гибкость:** Контекстные данные позволяют динамически изменять содержимое шаблона в зависимости от запроса.
- **Повышенная производительность:** Внутренне Django кэширует шаблоны, что снижает накладные расходы при их рендеринге.



```
1 from django.shortcuts import render
2
3 def index(request):
4     context = {'title': 'Главная страница', 'content': 'Добро пожаловать на наш сайт!'}
5     return render(request, 'index.html', context)
```

Рисунок 20 - Пример использования функции render

В шаблоне about.html данные контекста могут быть использованы следующим образом:



```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>{{ title }}</title>
5  </head>
6  <body>
7      <h1>{{ content }}</h1>
8      <h2>Наша команда</h2>
9      <ul>
10         {% for member in team_members %}
11             <li>{{ member }}</li>
12         {% endfor %}
13     </ul>
14  </body>
15  </html>
```

Рисунок 21 - Шаблон about.html

Использование функции render в Django упрощает процесс разработки, делая его более интуитивно понятным и удобным. Благодаря возможности передачи контекста, разработчики могут легко динамически изменять содержимое шаблонов, что позволяет создавать более интерактивные и адаптивные веб-приложения.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Для этой практической нужно воспользоваться проектом из первой практической работы

1. Задаем в основной файл urls.py, которая находится в директории app, и импортируем функцию include. После этих действий добавляем в переменную urlpatterns новый путь с помощью path.
(перед этим удалите прошлый путь)
Вторым аргументом в функции path, напишите include('hello.urls')



```
1 from django.contrib import admin
2 from django.urls import path, include
3
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6     path('', include('hello.urls')),
7 ]
8
```

Рисунок 22 - Основной urls.py

2. В приложении hello создаем файл urls.py, и заполняем его



```
1 from django.urls import path
2 from .views import get_hello
3
4 urlpatterns = [
5     path('', get_hello)
6 ]
```

Рисунок 23 - Файл urls.py в директории hello

3. Запускаем сервер для проверки работоспособности

Hello world!

Рисунок 24 - Веб-страница Hello world!

4. Теперь создаем директорию templates(**обязательно так**) в приложение hello, затем в папке templates создаем html файл(например hello.html).

Заполните html файл:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>{{title}}</title>
7  </head>
8  <body>
9      <h1> страница {{title}}</h1>
10 </body>
11 </html>
```

Рисунок 25 - Файл hello.html

5. В файле view.py импортируем функцию render для использование html файлов, потом меняем функцию get_hello. В функции render третьем аргументам передаем данные, которые используется для отображение переменных в html шаблоне



```
1 from django.shortcuts import render
2
3
4 def get_hello(request):
5     return render(request, 'hello.html',{
6         'title': 'Hello',
7     })
```

Рисунок 26 - Функция get_hello

6. Запускаем сервер!! И вам выдает ошибку, это ошибка происходит потому что приложение hello не подключено к основному проекту. Чтобы подключить приложение hello к основному проекту, заходим в файл settings.py, и в переменную INSTALLED_APPS



```
1 INSTALLED_APPS = [
2     'django.contrib.admin',
3     'django.contrib.auth',
4     'django.contrib.contenttypes',
5     'django.contrib.sessions',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'hello',
9 ]
10
```

Рисунок 27 - Переменная INSTALLED_APPS в файле settings.py

7. Теперь запуск сервера не выдает ошибку

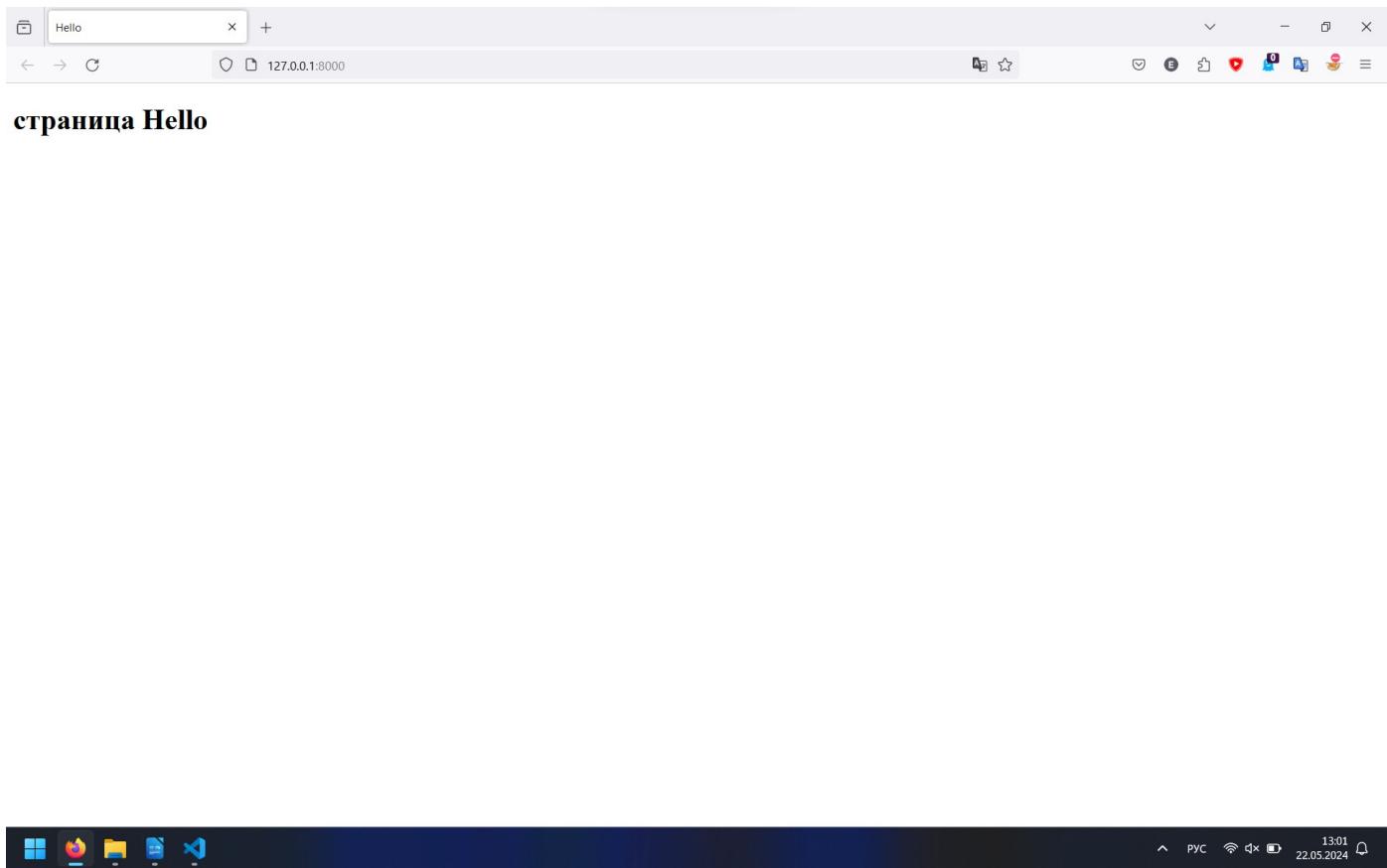


Рисунок 28 - Веб-страница hello

Задание

1. Создайте приложение library и подключите его
2. Выведите какуюнибудь книгу с помощью переменной по ссылке library/library

Контрольные вопросы

1. В каком файле подключаем приложение в django?
2. Для чего нужна функция render?
3. В какой папке сохраняем html шаблоны?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №3

Создание моделей

ЦЕЛЬ ЗАНЯТИЯ

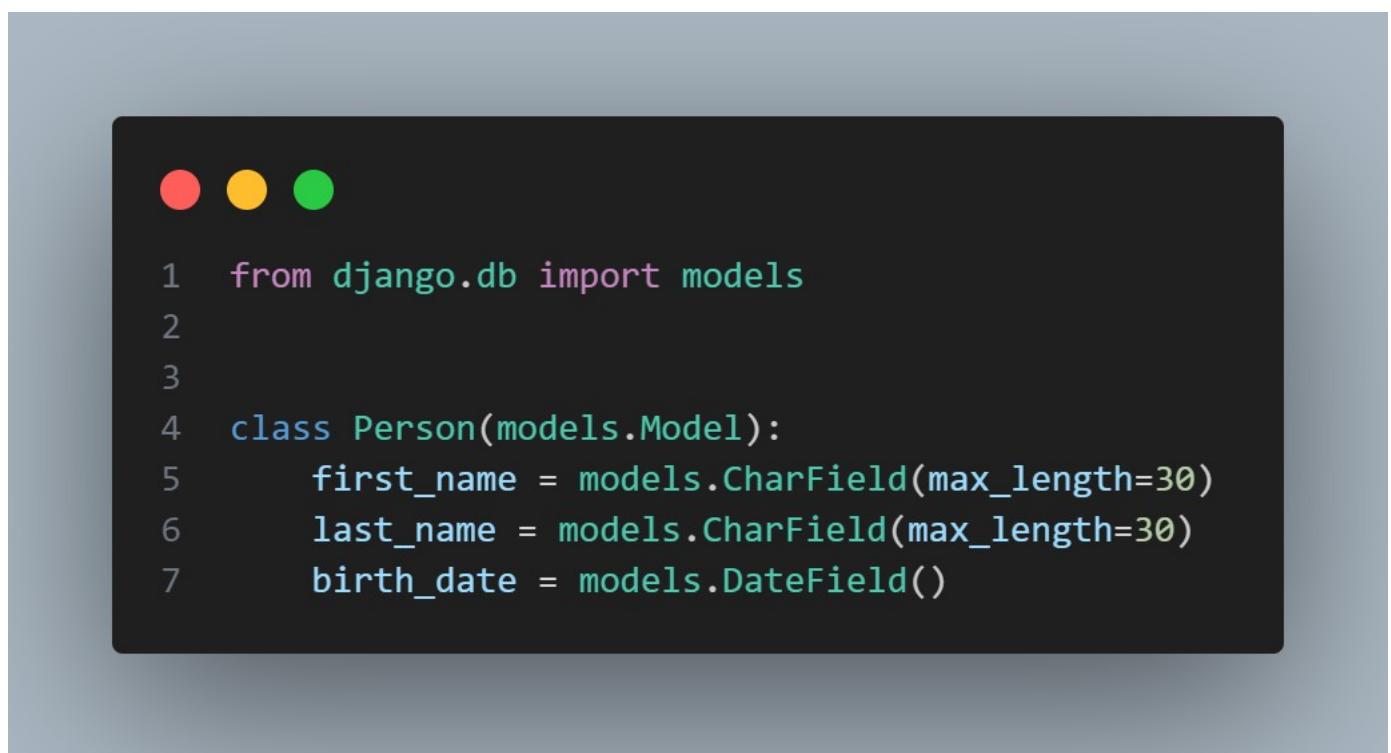
1. Познакомиться с моделями в Django
2. Научиться делать миграции моделей в БД

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Модель в Django представляет собой структуру, которая определяет форму данных в приложении. Она является отображением таблицы в базе данных и определяет поля и их типы, а также различные ограничения и связи между таблицами.

Основные компоненты моделей

1. **Класс модели:** В Django каждая модель представляется в виде класса, который наследуется от `django.db.models.Model`



```
● ● ●
1 from django.db import models
2
3
4 class Person(models.Model):
5     first_name = models.CharField(max_length=30)
6     last_name = models.CharField(max_length=30)
7     birth_date = models.DateField()
```

Рисунок 29 -Файл `models.py` с моделью `Person`

- **Поля модели:** Поля модели определяют тип данных, которые будут храниться в каждом столбце таблицы базы данных. Некоторые из распространенных типов полей включают:
 - CharField: для хранения строк.
 - IntegerField: для хранения целых чисел.
 - DateField: для хранения дат.

- ForeignKey: для создания связи многие-к-одному.
- **Метаданные модели:** Класс Meta внутри модели позволяет указать дополнительные параметры модели, такие как порядок сортировки, имя таблицы в базе данных и др.



```
1  from django.db import models
2
3
4  class Person(models.Model):
5      first_name = models.CharField(max_length=30)
6      last_name = models.CharField(max_length=30)
7      birth_date = models.DateField()
8
9      class Meta:
10         ordering = ['last_name']
11         db_table = 'person'
```

Рисунок 30 -Добавление метакласса

Создание и работа с моделями

1. **Создание моделей:** Модели создаются путем написания классов в файле models.py вашего приложения. После создания модели необходимо выполнить миграции для применения изменений в базу данных.
2. **Команды makemigrations и migrate:**
 - **makemigrations:** Эта команда анализирует изменения в моделях и создает файл миграции, который фиксирует эти изменения. Миграции – это инструкции, которые Django использует для внесения изменений в структуру базы данных.

```
python manage.py makemigrations
```

Пример вывода команды:

```
Migrations for 'myapp':
myapp/migrations/0001_initial.py:
- Create model Person
```

Файл миграции будет содержать инструкции для создания таблицы Person в базе данных.

- **migrate:** Эта команда применяет созданные миграции к базе данных, выполняя соответствующие SQL-запросы для изменения структуры базы данных.

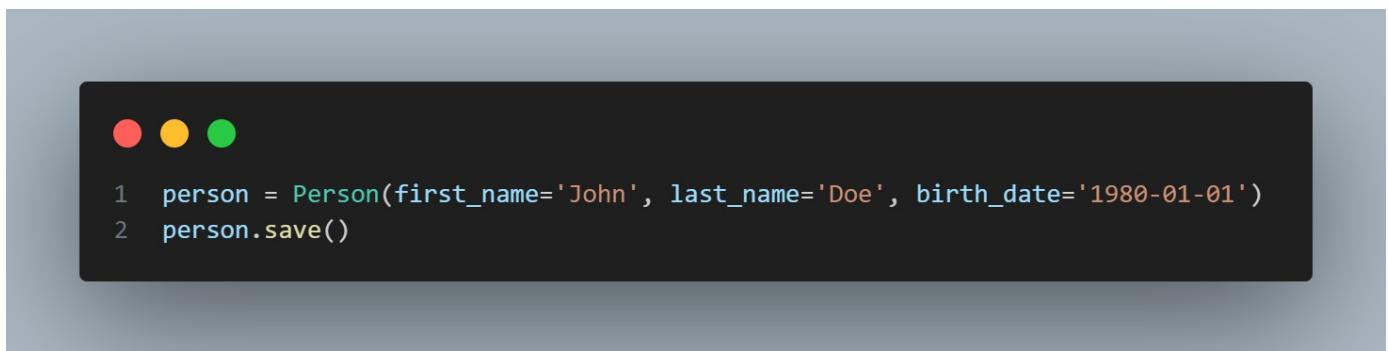
```
python manage.py migrate
```

Пример вывода команды:

- **Applying myapp.0001_initial... OK**

После выполнения команды migrate таблицы и другие структуры данных будут созданы в базе данных.

- **Создание и сохранение объектов:** Объекты модели создаются и сохраняются с помощью методов, предоставляемых Django.

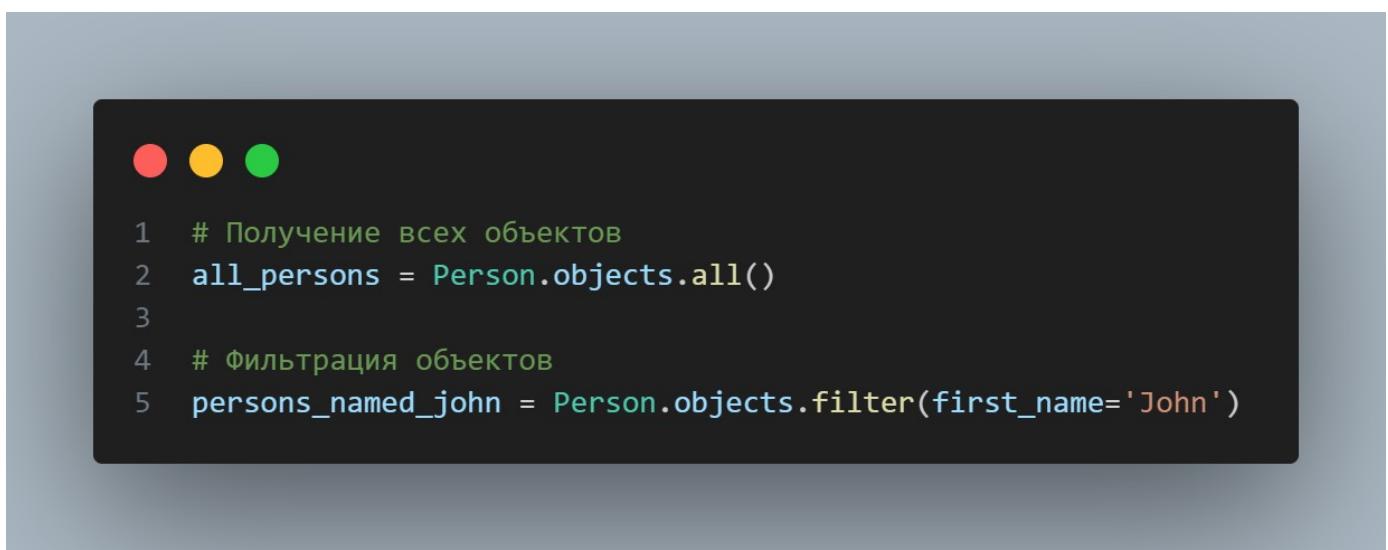


```
1 person = Person(first_name='John', last_name='Doe', birth_date='1980-01-01')
2 person.save()
```

Рисунок 31 -Заполнение базы данных с помощью ORM

- **Запросы к базе данных:** Django предоставляет мощный API для выполнения запросов к базе данных.

```
python
```



```
1 # Получение всех объектов
2 all_persons = Person.objects.all()
3
4 # Фильтрация объектов
5 persons_named_john = Person.objects.filter(first_name='John')
```

Рисунок 32 - Получение данных из БД с помощью ORM

- **Обновление и удаление объектов:** Django позволяет легко обновлять и удалять объекты.



```
1 # Обновление объекта
2 person = Person.objects.get(id=1)
3 person.last_name = 'Smith'
4 person.save()
5
6 # Удаление объекта
7 person.delete()
```

Рисунок 33 - Изменение и удаление данных из БД

Модели Django – это мощный инструмент для работы с данными в веб-приложениях. Они предоставляют удобный и интуитивно понятный способ определения структуры данных и выполнения операций с ними. Понимание основ работы с моделями является ключевым навыком для любого разработчика, работающего с Django. Команды makemigrations и migrate помогают эффективно управлять изменениями в структуре данных, обеспечивая синхронизацию моделей и базы данных.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Для этой практической нужно скачать расширение для VS code - SQLite Viewer.



Рисунок 34 - Расширение для VS code

Или скачать [SQLiteStudio](#) для просмотра данных в БД.

1. Создайте проект и приложение
2. Запускаем сервер
3. Заходим в файл models.py в вашем приложении. Создаем модель Book:

A screenshot of a code editor showing a Python file named "models.py". The code defines a "Book" model using Django's "Model" class. It includes two fields: "name" and "author", both defined using "CharField".

```
1 class Book(models.Model):
2     name = models.CharField()
3     author = models.CharField()
```

Рисунок 35 - Модель Book

После создание модели у вас должно было вывести ошибку в терминал(если у вас запущен сервер)

A screenshot of a terminal window showing a Django model validation error. The error message indicates that the "CharFields" for "book.Book.name" and "book.Book.author" must define a "max_length" attribute.

```
ERRORS:
book.Book.author: (fields.E120) CharFields must define a 'max_length' attribute.
book.Book.name: (fields.E120) CharFields must define a 'max_length' attribute.
```

Рисунок 36 - Ошибка терминали

Эта ошибка вызывается из за незаполненного атрибута класса **CharField**.

Как понять какой атрибут нужно заполнить? Обычно Django сам пишет что нужно заполнить в терминале. В данном примере нужно заполнить атрибут `max_length` у класса **CharField**.



```
1 from django.db import models
2
3
4 class Book(models.Model):
5     name = models.CharField(max_length=255)
6     author = models.CharField(max_length=10)
```

Рисунок 37 - Модель с добавленными атрибутами

4. Пишем в терминал:

```
py manage.py makemigrations
```

После выполнение команды, в терминале появиться сообщение об успешном создание файла миграции:

```
Migrations for 'book':
  book\migrations\0001_initial.py
    - Create model Book
```

Рисунок 38 - Сообщение об создание файла для миграции

В папке migrations находим файл 0001_initial.py с файлом миграцией.

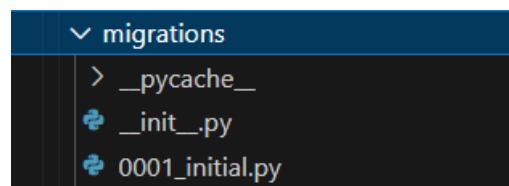


Рисунок 39 - Папка migrations

```
● ● ●
1  from django.db import migrations, models
2
3
4  class Migration(migrations.Migration):
5      initial = True
6
7      dependencies = [
8          ]
9
10     operations = [
11         migrations.CreateModel(
12             name='Book',
13             fields=[
14                 ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
15                 ('name', models.CharField(max_length=255)),
16                 ('author', models.CharField(max_length=10)),
17             ],
18         ),
19     ]
20
21 
```

Рисунок 40 - Скрипт для миграций моделей в БД

В скрипте можно заменить, что добавилось еще одно поле id(PK - Primary Key – Первичный ключ). Поле id добавляется автоматический для каждой модели.

После создание скрипта для миграции надо вызвать этот скрипт.

5. Команда для миграции в БД:

py manage.py migrate

```
Operations to perform:
  Apply all migrations: admin, auth, book, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying book.0001_initial... OK
  Applying sessions.0001_initial... OK
```

Рисунок 41 - Миграции моделей в БД

После написания команды в терминале выводиться сообщение с файлами миграции.(Например на рисунке - Миграции моделей в БД создалась таблица book(предпоследняя строчка))

Можно заметить, что очень много совершилось миграции в БД. Это [встроенные модели](#) django, которые были написаны для быстрого создания веб-приложений, например модель User.

Откройте файл db.sqlite3 с помощью VS CODE и расширения SQLite Viewer

	id	name	author
Filter...	=	Filter...	=

Рисунок 42 - Файл db.sqlite3

Теперь в БД есть таблица book.

Задание

Задание делаем в проекте из практической части

1. Создайте модель Author с полями name, date_of_birth, gender и добавьте в БД
2. Добавьте модель в book, поле year

(Чтобы добавить поле year, повторите действие с добавление модели в БД)

Контрольные вопросы

1. Для чего нужен файл models.py?
2. Какая команда создает скрипт для миграции ?
3. В какую папке хранятся скрипты для миграции?
4. Какая команда активирует скрипты миграции ?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №4

Связи

ЦЕЛЬ ЗАНЯТИЯ

1. Научиться создавать связи в моделях

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

В этой теоретической части мы рассмотрим связи между моделями в Django, а именно: один-к-одному, один-ко-многим и многие-ко-многим.

Основные типы связей

Один-к-одному (One-to-One)

Связь типа один-к-одному означает, что каждой записи одной модели соответствует одна и только одна запись другой модели. В Django это реализуется с помощью OneToOneField.

Пример:

```
1 from django.db import models
2
3 class UserProfile(models.Model):
4     user = models.OneToOneField(User, on_delete=models.CASCADE)
5     bio = models.TextField()
6
7 # В этом примере у каждой записи User будет ровно одна запись UserProfile.
```

Рисунок 43 -Один к одному

Один-ко-многим (One-to-Many)

Это наиболее распространенный тип связи, где одна запись одной модели может быть связана с множеством записей другой модели. В Django это реализуется с помощью ForeignKey.

Пример:



```
1 from django.db import models
2
3 class Author(models.Model):
4     name = models.CharField(max_length=100)
5
6 class Book(models.Model):
7     title = models.CharField(max_length=200)
8     author = models.ForeignKey(Author, on_delete=models.CASCADE)
9
10 # В этом примере один автор может написать много книг.
```

Рисунок 44 - Один-ко-многим

Многие-ко-многим (Many-to-Many)

Связь многие-ко-многим означает, что каждая запись одной модели может быть связана с множеством записей другой модели и наоборот. В Django это реализуется с помощью ManyToManyField.

Пример:



```
1 from django.db import models
2
3 class Student(models.Model):
4     name = models.CharField(max_length=100)
5
6 class Course(models.Model):
7     title = models.CharField(max_length=200)
8     students = models.ManyToManyField(Student)
9
10 # В этом примере один студент может записаться на много курсов, и один курс может быть посещаем многими студентами.
```

Рисунок 45 - Многие ко многим

Опции связи

При определении связей в Django, важно учитывать несколько опций:

on_delete — определяет поведение при удалении связанных объектов:

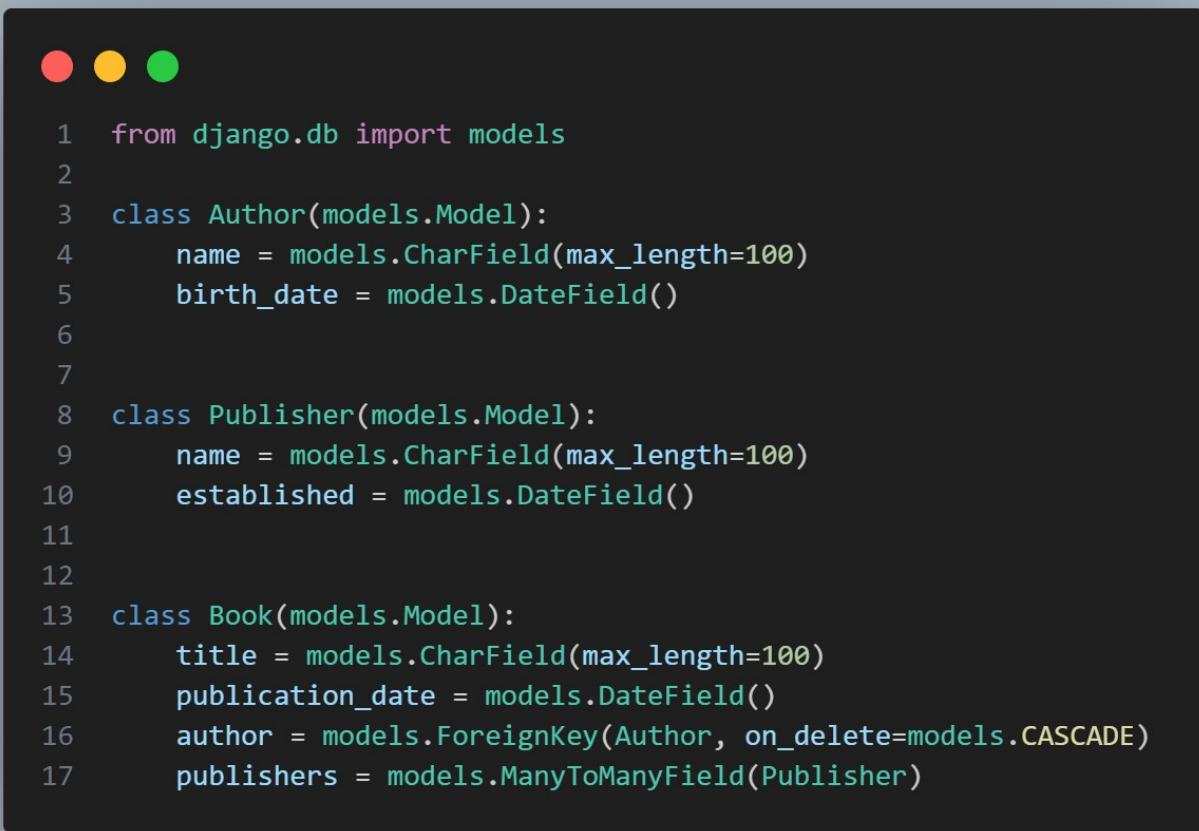
- CASCADE: при удалении записи родительской модели, удаляются и все связанные записи.
- PROTECT: предотвращает удаление родительской записи, если существуют связанные записи.
- SET_NULL: устанавливает значение NULL в связанных записях.
- SET_DEFAULT: устанавливает значение по умолчанию в связанных записях.

- DO NOTHING: ничего не делать.
- related_name** — задает имя обратной связи для доступа к связанным записям.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Для этой практической создайте новый проект с приложением library

1. Заходим в файл models.py. Создаем модели Author,Publisher, Book



```
1 from django.db import models
2
3 class Author(models.Model):
4     name = models.CharField(max_length=100)
5     birth_date = models.DateField()
6
7
8 class Publisher(models.Model):
9     name = models.CharField(max_length=100)
10    established = models.DateField()
11
12
13 class Book(models.Model):
14     title = models.CharField(max_length=100)
15     publication_date = models.DateField()
16     author = models.ForeignKey(Author, on_delete=models.CASCADE)
17     publishers = models.ManyToManyField(Publisher)
```

Рисунок 46 - Файл models.py с моделями Author, Publisher, Book

(Делаем миграцию в БД)

2. Заходим в БД, и видим что создалось 4 таблице. Это происходит из за связи многим ко многим. Django автоматически создает таблицу со связями.

```
> auth_group
> auth_group_permissions
> auth_permission
> auth_user
> auth_user_groups
> auth_user_user_permissions
> book_author
> book_book
> book_book_publishers
> book_publisher
> django_admin_log
> django_content_type
> django_migrations
> django_session
> sqlite_sequence
```

Рисунок 47 - Таблицы в БД

3. Теперь добавим организацию импортов. Для этого удаляем файл models.py и создаем папку models. В папке models создаем файл __init__.py, затем создаем под каждую модель свой файл.

Например:

```
1 from django.db import models
2
3 class Publisher(models.Model):
4     name = models.CharField(max_length=100)
5     established = models.DateField()
```

Рисунок 48 - Файл publisher.py

4. После создание всех файлов для моделей, заполняем файл __init__.py



```
1  from .author import Author
2  from .book import Book
3  from .publisher import Publisher
4
5  __all__ =(
6      'Author',
7      'Book',
8      'Publisher',
9
10 )
11 )
```

Рисунок 49 - Файл `__init__.py`

5. Теперь при импорте моделей в другой файл, можно не указывать полный путь до файла модели

Задание

1. Создайте модели Student, Course и Teacher с соответствующими полями и связями:
2. Связь многие-ко-многим между Student и Course.
3. Связь один-ко-многим между Teacher и Course.

Контрольные вопросы

1. Что такое связь один-к-одному (One-to-One) и как она реализуется в Django?
2. Что такое связь один-ко-многим (ForeignKey) и как она реализуется в Django?
3. Что такое связь многие-ко-многим (Many-to-Many) и как она реализуется в Django?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №5

Админ-панель

ЦЕЛЬ ЗАНЯТИЯ

1. Создание суперпользователя
2. Добавление моделей в админ панель

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Что такое админ панель в Django?

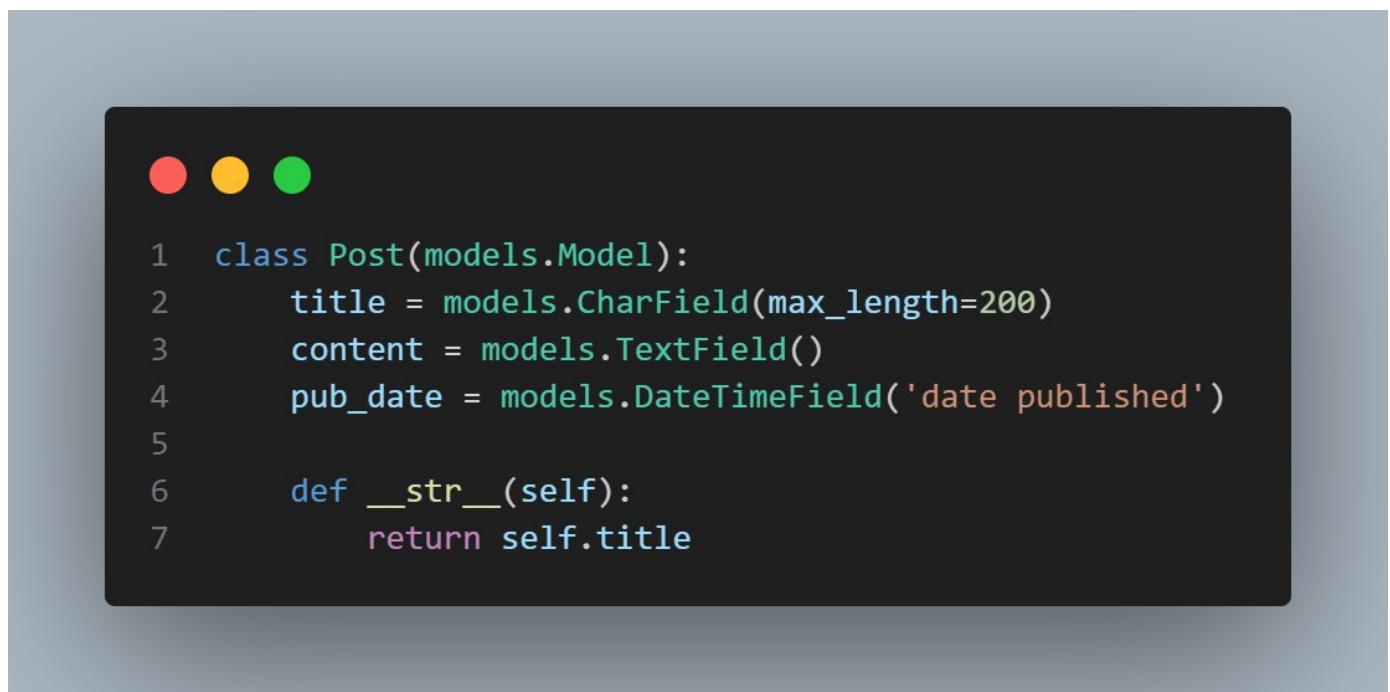
Админ панель Django - это встроенный интерфейс, предназначенный для управления моделями данных. Она позволяет администраторам сайта добавлять, изменять и удалять данные через удобный веб-интерфейс. Админ панель является важным инструментом, который ускоряет разработку и облегчает управление контентом.

Основные компоненты админ панели

1. **Модели (Models):** В Django админ панель работает с моделями данных. Модели определяются в файлах models.py каждого приложения и представляют собой структуру данных, которая сохраняется в базе данных.
2. **Административные классы (Admin classes):** Для настройки админ панели используются административные классы, которые определяются в файле admin.py. Эти классы позволяют настроить отображение и функциональность административного интерфейса.
3. **Регистрация моделей:** Чтобы модель появилась в админ панели, её нужно зарегистрировать. Это делается в файле admin.py с помощью декораторов.

Настройка админ панели

1. **Создание модели:** Прежде чем настроить админ панель, нужно создать модели. Пример модели:



```
1 class Post(models.Model):
2     title = models.CharField(max_length=200)
3     content = models.TextField()
4     pub_date = models.DateTimeField('date published')
5
6     def __str__(self):
7         return self.title
```

Рисунок 50 - Модель post

- **Регистрация модели в админ панели с использованием декоратора:** После создания модели её нужно зарегистрировать в админ панели. Это делается в файле admin.py с использованием декоратора @admin.register():

```
● ● ●  
1 from django.contrib import admin  
2 from .models import Post  
3  
4 @admin.register(Post)  
5 class PostAdmin(admin.ModelAdmin):  
6     list_display = ('title', 'pub_date')  
7     search_fields = ('title',)
```

Рисунок 51 - Добавление модель в админ панель

Расширенные настройки админ панели

1. **Настройка форм:** Можно настроить, какие поля будут отображаться и в каком порядке:

```
● ● ●  
1 @admin.register(Post)  
2 class PostAdmin(admin.ModelAdmin):  
3     fields = ['title', 'pub_date', 'content']
```

Рисунок 52 - Отображение полей для админ панели

Фильтрация данных: Для удобства можно добавить фильтры по полям:



```
1 @admin.register(Post)
2 class PostAdmin(admin.ModelAdmin):
3     list_filter = ['pub_date']
```

Рисунок 53 - Добавление фильтрации по полю

Редактирование связанных объектов: Админ панель позволяет редактировать связанные объекты с помощью inline-форм:



```
1 from django.contrib import admin
2 from .models import Post, Comment
3
4
5 class CommentInline(admin.TabularInline):
6     model = Comment
7
8 @admin.register(Post)
9 class PostAdmin(admin.ModelAdmin):
10    inlines = [CommentInline]
```

Рисунок 54 - Добавление связанных моделей

Безопасность админ панели

Админ панель является мощным инструментом и требует надлежащей защиты. Вот несколько рекомендаций по обеспечению безопасности админ панели:

- Использование HTTPS:** Всегда используйте защищённое соединение для доступа к админ панели.
- Ограничение доступа:** Настройте доступ к админ панели только для доверенных пользователей и IP-адресов.
- Регулярное обновление:** Убедитесь, что ваше приложение и все зависимости всегда обновлены до последних версий.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Для этой практической части возьмите проект из прошлой практической части.

1. Создаем суперпользователя для входа в админ панель. Пишем в терминал:

py manage.py createsuperuser

После ввода команды в терминале появиться регистрация для суперпользователя.

При вводе пароля пароль не будет отображаться

Пример:

```
Username: admin
Email address: admin@gmail.com
Password:
Password (again):
The password is too similar to the username.
This password is too short. It must contain at least 8 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Рисунок 55 - Регистрация суперпользователя через терминал

2. Запускаем сервер, и заходим по url – admin. После захода, вас перекинет на страницу авторизации админа, вводим данные, которые использовали при регистрации.



Рисунок 56 - Страница авторизации суперпользователя

3. Если авторизация прошла успешно, то вас перекинет в админ-панель.

The image shows a screenshot of the Django admin panel. The top navigation bar includes 'WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT'. The main content area has a header 'AUTHENTICATION AND AUTHORIZATION'. Below it, there are two sections: 'Groups' and 'Users'. Under 'Groups', there are '+ Add' and 'Change' buttons. Under 'Users', there are '+ Add' and 'Change' buttons. To the right, there is a sidebar with 'Recent actions' and 'My actions' sections, both of which say 'None available'. The overall layout is clean and organized, typical of the Django admin interface.

Рисунок 57 - Страница админ-панели

В админ-панель уже есть некоторые модели, которые встроенные в Django, это модели User, Group.

4. Теперь добавляем модели в админ-панель. Для этого заходим в файл admin.py. Импортируем модели Author, Book, Publisher, затем создаем класс для настройки страницы модели в админ-панели(класс должен наследовать класс ModelAdmin).

Регистрируем модель в этой классе. Для этого нужно использовать декоратор register в котором записываем модель



```
1 from django.contrib import admin
2 from book.models import Book, Publisher, Author
3
4
5 @admin.register(Author)
6 class AdminAuthor(admin.ModelAdmin):
7     pass
8
```

Рисунок 58 - Добавление модели в админ-панель

5. Заходим в админ-панель. Теперь в админ-панели появилась модель Author. Добавьте каких нибудь книг автором, чтобы проверить работоспособность админ-панели.
6. Заходим в db.sqlite3 для проверить загрузки данных в БД

The screenshot shows the DB Browser for SQLite interface. On the left, there's a tree view of tables under the 'app > db.sqlite3' connection. The 'book_author' table is selected and highlighted with a blue border. On the right, there's a grid view of the 'book_author' table with one row displayed. The columns are labeled 'id', 'name', and 'birth_date'. The first row has values: id=1, name='Александр Сергеевич Пушкин', and birth_date='1799-06-06'.

		id	name	birth_date
		1	Александр Сергеевич Пушкин	1799-06-06

Рисунок 59 - Таблица book_author

Задание

1. Добавьте модели Book и Publisher в админ-панель.
2. Заполните таблица book, publisher, author.

Контрольные вопросы

1. В каком файле добавляется модель в админ-панели?
2. С помощью какого декоратора регистрируем модель в админ-панели?
3. Зачем нужна админ-панели?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №6

Django ORM запросы

ЦЕЛЬ ЗАНЯТИЯ

1. Создание запросов

2. Фильтрация запросов

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Запросы в ORM Django представляют собой удобный и мощный способ взаимодействия с базой данных. Они позволяют выполнять операции создания, чтения, обновления и удаления (CRUD) записей с помощью Python-кода, абстрагируя сложность SQL-запросов.

Основные типы запросов

Создание записей

Создание записей в базе данных осуществляется с использованием метода `create()` или через экземпляры моделей и метод `save()`.



```
● ● ●
1 # Использование метода create()
2 author = Author.objects.create(name='George Orwell', birth_date='1903-06-25')
3
4 # Создание экземпляра модели и использование метода save()
5 book = Book(title='1984', author=author, published_date='1949-06-08', price=19.99)
6 book.save()
```

Рисунок 60 - Создание записей

Чтение данных

ORM Django предоставляет различные методы для выборки данных из базы. Основные из них включают: `all()`, `get()`, `filter()`, `exclude()`, `order_by()` и `values()`.



```
1 # Получение всех записей
2 all_authors = Author.objects.all()
3
4 # Получение одной записи по уникальному критерию
5 specific_author = Author.objects.get(id=1)
6
7 # Фильтрация записей по условию
8 books_by_orwell = Book.objects.filter(author__name='George Orwell')
9
10 # Исключение записей по условию
11 books_not_by_orwell = Book.objects.exclude(author__name='George Orwell')
12
13 # Сортировка записей
14 books_ordered_by_date = Book.objects.order_by('published_date')
15
16 # Выбор определенных полей
17 book_titles = Book.objects.values('title')
```

Рисунок 61 - Чтение записей

Обновление данных

Для обновления данных используется метод `save()` после изменения атрибутов модели или метод `update()` для массового обновления.



```
1 # Обновление через экземпляр модели и метод save()
2 author = Author.objects.get(id=1)
3 author.name = 'Eric Arthur Blair'
4 author.save()
5
6 # Массовое обновление через метод update()
7 Book.objects.filter(author__name='George Orwell').update(price=20.99)
8
```

Рисунок 62 - Обновление данных

Удаление данных

Удаление данных осуществляется с использованием метода `delete()`.

```
● ● ●  
1 # Удаление одного экземпляра  
2 author = Author.objects.get(id=1)  
3 author.delete()  
4  
5 # Массовое удаление через фильтр  
6 Book.objects.filter(author__name='George Orwell').delete()  
7
```

Рисунок 63 - Удаление данных

Комплексные запросы

Связи между моделями

Django ORM поддерживает запросы через связанные модели, используя двойное подчеркивание (`__`) для обращения к полям связанных моделей.

```
● ● ●  
1 # Получение всех книг определенного издателя  
2 publisher_books = Book.objects.filter(publisher__name='Penguin Books')  
3  
4 # Получение всех авторов книг, изданных после 2000 года  
5 authors_of_recent_books = Author.objects.filter(book__published_date__gt='2000-01-01')
```

Рисунок 64 - Запросы по связям

Агрегирование данных

Django ORM поддерживает агрегатные функции, такие как `Count`, `Avg`, `Max`, `Min`, и `Sum`.



```
1 from django.db.models import Count, Avg  
2  
3 # Подсчет количества книг у каждого автора  
4 author_book_count = Author.objects.annotate(book_count=Count('book'))  
5  
6 # Средняя цена книг  
7 average_book_price = Book.objects.aggregate(Avg('price'))
```

Рисунок 65 - Агрегирование данных

Аннотации

Аннотации позволяют добавлять вычисляемые поля к запросам.



```
1 from django.db.models import F  
2  
3 # Вычисление скидочной цены книг  
4 books_with_discount = Book.objects.annotate(discounted_price=F('price') * 0.9)
```

Рисунок 66 - Аннотации

Управление транзакциями

Django ORM предоставляет инструменты для управления транзакциями, что позволяет гарантировать целостность данных.

```
1 from django.db import transaction
2
3 @transaction.atomic
4 def create_book_and_author():
5     author = Author.objects.create(name='New Author', birth_date='1980-01-01')
6     Book.objects.create(title='New Book', author=author, published_date='2023-01-01', price=29.99)
```

Рисунок 67 - Управление транзакциями

Оптимизация запросов

Для повышения производительности запросов в Django используются методы `select_related()` и `prefetch_related()`, которые уменьшают количество запросов к базе данных.

```
1
2 # Использование select_related для жадной загрузки
3 books_with_authors = Book.objects.select_related('author').all()
4
5 # Использование prefetch_related для ленивой загрузки
6 books_with_publishers = Book.objects.prefetch_related('publisher').all()
```

Рисунок 68 - Оптимизация запросов

Запросы в ORM Django предоставляют мощные и гибкие инструменты для взаимодействия с базой данных. Используя ORM, разработчики могут эффективно выполнять различные операции с данными, обеспечивая высокую читаемость и поддерживаемость кода. ORM Django абстрагирует сложность SQL-запросов, позволяя сосредоточиться на логике приложения.

ПРАКТИЧЕСКАЯ ЧАСТЬ

В этой практической будет использовать только чтение БД

1. Возьмите проект из прошлой практической
2. Заполните базу данных с помощью админ-панели
3. Откройте файл views.py, и импортируйте модель Book
4. Создай функцию для отображения html файла. В функции создаем переменную book, которой будет храниться запись под id=1 в таблице book. Для чтения этой таблицы надо использовать Django ORM. Используем модель Book. Пишем Book.objects.get(любое_поле_из_модели = Значение)



```
1 def get_info(request):
2     book = Book.objects.get(id=1)
3     return render(request, 'base.html', context={
4         'book' : book,
5     })
```

Рисунок 69 - Функция для отображения html файла

5. Создаем и пишем в файле base.html



```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Document</title>
7 </head>
8 <body>
9     {{ book.title }}
10    {{ book.author.name }}
11 </body>
12 </html>
```

Рисунок 70 - Файл base.html

В html файле используем в переменную, которую передали в функций render.

В переменной храниться объект класса Book с данными из записи.

Рисунок 71 - Веб-страница с данными из БД

7. Создаем еще одну функцию для отображения. В которой используем метод filter вместо get, с помощью фильтр можно взять несколько запись вместо одного(как в get).
8. Вместо поиска по одному полю из таблицы, будем искать по связи поля author.

Для этого пишем в filter аргументы, которые используются в модели, например:



```
1 def get_author(request):
2     book_author = Book.objects.filter(author__name="Александр Сергеевич Пушкин")
3     return render(request, 'author.html', context= {
4         'book_author': book_author,
5     })
```

Рисунок 72 - Фильтрация по полю author

Теперь в переменной book_author храниться запись из таблицы book с автором Пушкин

9. Создаем файл author.html, заполняем html разметку и используем шаблонизатор для отображение всех книг с автором Пушкин. В шаблонизаторе можно использовать цикл for для итерации переменной, которую передали.



```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7  </head>
8  <body>
9      {% for i in book_author %}
10         <h1>{{ i.title }}</h1>
11     {% endfor %}
12  </body>
13 </html>
```

Рисунок 73 - Html файл с использованием шаблонизатора

10. Подключите функцию к urls. Зайдите на страницу по urls, теперь на страницу будут показывать все книги с автором Пушкин.

Евгений Онегин

Борис Годунов

Рисунок 74 - Веб-страница с перечислением авторов

Задание

Для этой работы заполните данные в админ-панели

1. Создайте функцию отображение, которая выводит данные из связи Author «один ко многим»(Кроме Пушкина. Выводить по любому значению.)
2. Создайте функцию отображение, которая выводит данные из связи Publisher «многим ко многим»(Выводить по любому значению)

Контрольные вопросы

1. С чем отличает метод get от filter?
2. Как создается запрос по связям?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №7

Шаблонизатор в Django, статические файлы

ЦЕЛЬ ЗАНЯТИЯ

1. Подключение статических файлов
2. Использование шаблонов

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Шаблоны в Django — это текстовые файлы, которые могут содержать статический HTML и динамические конструкции, такие как переменные и теги. Они позволяют динамически формировать HTML на основе данных, переданных из представлений.

Основные элементы шаблонизатора

Переменные: Переменные используются для вывода данных, переданных из представления. В шаблоне переменные обрамляются двойными фигурными скобками.

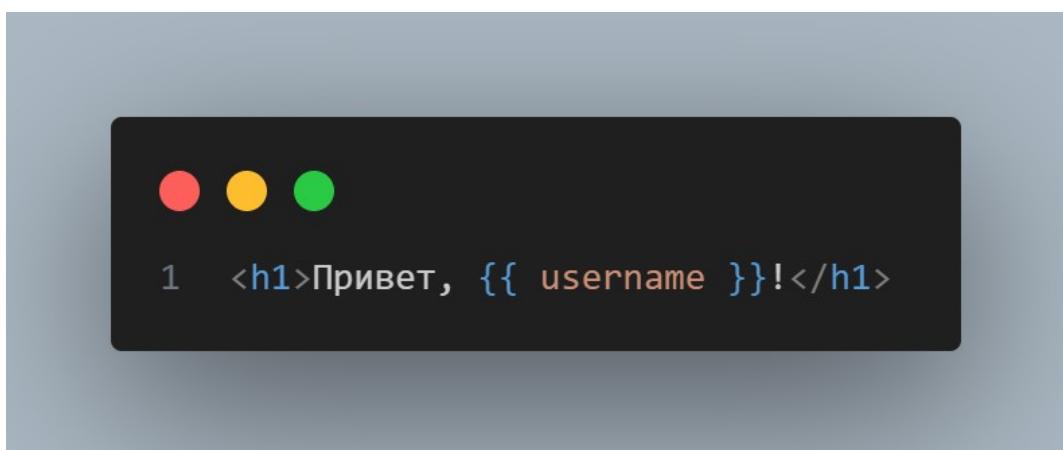


Рисунок 75 - Использование переменных в шаблонизаторе

Теги:

Теги — это более сложные конструкции, которые могут включать условия, циклы, подгрузку других шаблонов и многое другое. Они обрамляются фигурными скобками с процентами.

Условия:



```
1  {% if user.is_authenticated %}  
2      <p>Привет, {{ user.username }}!</p>  
3  {% else %}  
4      <p>Привет, гость!</p>  
5  {% endif %}
```

Рисунок 76 - Условия в шаблонизаторе

Циклы:



```
1  <ul>  
2      {% for item in item_list %}  
3          <li>{{ item }}</li>  
4      {% endfor %}  
5  </ul>
```

Рисунок 77 - Цикл *for* в шаблонизаторе

Подключение шаблонов:



Рисунок 78 - Подключение шаблонов

Шаблонизатор Django и работа со статическими файлами являются важными аспектами при разработке веб-приложений. Шаблонизатор позволяет эффективно отделять логику от представления, а статические файлы обеспечивают необходимую функциональность и внешний вид вашего сайта. Используя эти инструменты, вы можете создавать динамичные, масштабируемые и поддерживаемые веб-приложения.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Для этой практической изменим архитектуру проекта

1. Заходим в файл settings.py
2. Импортируем библиотеку os. Находим переменную TEMPLATES, и на строку выше создаем переменную TEMPLATES_DIR со значением os.path.join(BASE_DIR, 'templates').

В этой переменной будет храниться путь к основной директории



```
1 TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

Рисунок 79 - Переменная TEMPLATES_DIR

После создание переменной, нужно заполнить в переменной TEMPLATES. В TEMPLATES есть ключ DIRS, нужно заполнить значение для этого ключа переменной TEMPLATES_DIR



```
1 TEMPLATES = [
2     {
3         'BACKEND': 'django.template.backends.django.DjangoTemplates',
4         'DIRS': [TEMPLATES_DIR],
5         'APP_DIRS': True,
6         'OPTIONS': {
7             'context_processors': [
8                 'django.template.context_processors.debug',
9                 'django.template.context_processors.request',
10                'django.contrib.auth.context_processors.auth',
11                'django.contrib.messages.context_processors.messages',
12            ],
13        },
14    },
15]
```

Рисунок 80 -Переменная TEMPLATES

3. Создаем папку templates в основной директории(не в приложении). Теперь папка templates будет основной для всех html файлов.

Например:

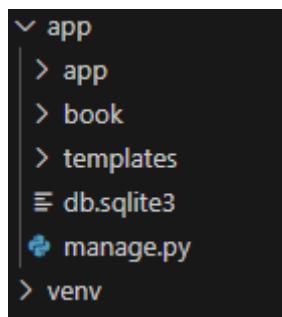


Рисунок 81 -
Архитектура
проекта

4. Подключим статические файлы (это файла, которые сервер не может изменить, например : js, css)

Заходим в settings.py. Ищем переменную STATIC_URL, и на строку ниже создаем переменную STATICFILES_DIRS со значение [os.path.join(BASE_DIR, 'static')]

```
1 STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

Рисунок 82 - Переменная STATICFILES_DIRS

Создаем папку static в основной директории.

5. Создаем базовый html файл base.html от которого будет наследоваться.



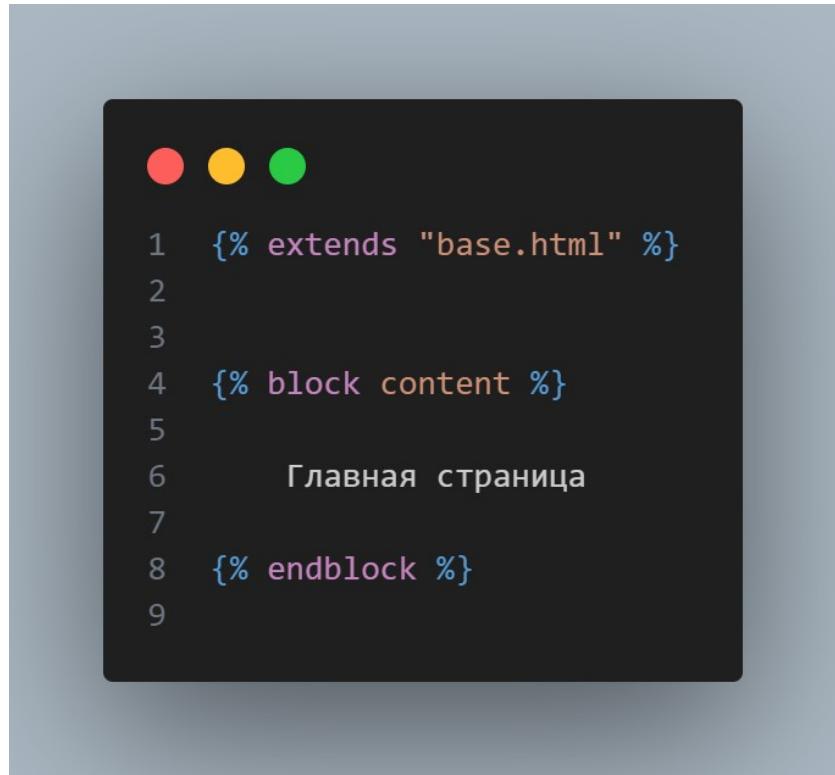
```
1  {% load static %}

2

3  <!DOCTYPE html>
4  <html lang="en">
5  <head>
6      <meta charset="UTF-8">
7      <meta name="viewport" content="width=device-width, initial-scale=1.0">
8      <link rel="stylesheet" href="{% static 'style/style.css' %}" >
9
10     <title>Document</title>
11 </head>
12 <body>
13
14     <header>
15         <div class="logo">
16
17             </div>
18         <div class="navigation">
19             <ul>
20                 <li><a href="">Главная страница</a></li>
21                 <li><a href="">0 нас </a></li>
22             </ul>
23         </div>
24
25     </header>
26
27
28     {% block content %}
29
30
31
32     {% endblock %}
33
34 </body>
35 </html>
```

Рисунок 83 - Файл base.html

6. Создаем еще один файл home.html



The screenshot shows a dark-themed code editor window. At the top, there are three colored circular icons: red, yellow, and green. Below them is the following Python template code:

```
1  {% extends "base.html" %}\n2\n3\n4  {% block content %}\n5\n6      Главная страница\n7\n8  {% endblock %}\n9
```

Рисунок 84 - Файл home.html

С помощью extends использует наследуемся от base.html. В content block пишем разметку которая будет отображаться в определенном месте(это можно увидеть в base.html)

7. Создай функцию для отображение файла home.html

- [Главная страница](#)
- [О нас](#)

Главная страница

Рисунок 85 - Веб-страница

Задание

1. Сделайте веб-приложение. Где будет главная страница, о нас и страница товаров. Тему выбирает по номеру из списка:

Дизайн должен соответствовать темы веб-приложения

- 1)Одежда и аксессуары
- 2)Электроника и гаджеты
- 3)Домашняя техника
- 4)Мебель и интерьер
- 5)Косметика и уход за кожей
- 6)Спортивные товары
- 7)Детские товары
- 8)Продукты питания
- 9)Книги и канцтовары
- 10)Автозапчасти и аксессуары
- 11)Сад и огород
- 12)Товары для животных
- 13)Хобби и творчество
- 14)Здоровье и фитнес
- 15)Игрушки и настольные игры
- 16)Туризм и отдых
- 17)Ювелирные изделия
- 18)Часы и оптика
- 19)Кухонная утварь
- 20)Музыкальные инструменты
- 21)Электротовары и освещение
- 22)Офисные товары
- 23)Антиквариат и коллекционные товары
- 24)Строительные материалы и инструменты
- 25)Ремесленные товары

Контрольные вопросы

1. Для чего нужен шаблонизатор ?
2. Как наследовать html шаблон?
3. Чем отличают {{}} от {%%}?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №8

Ссылки и медиафайл

ЦЕЛЬ ЗАНЯТИЯ

1. Создание ссылок
2. Сохранения медиафайлов

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Работа с ссылками в Django

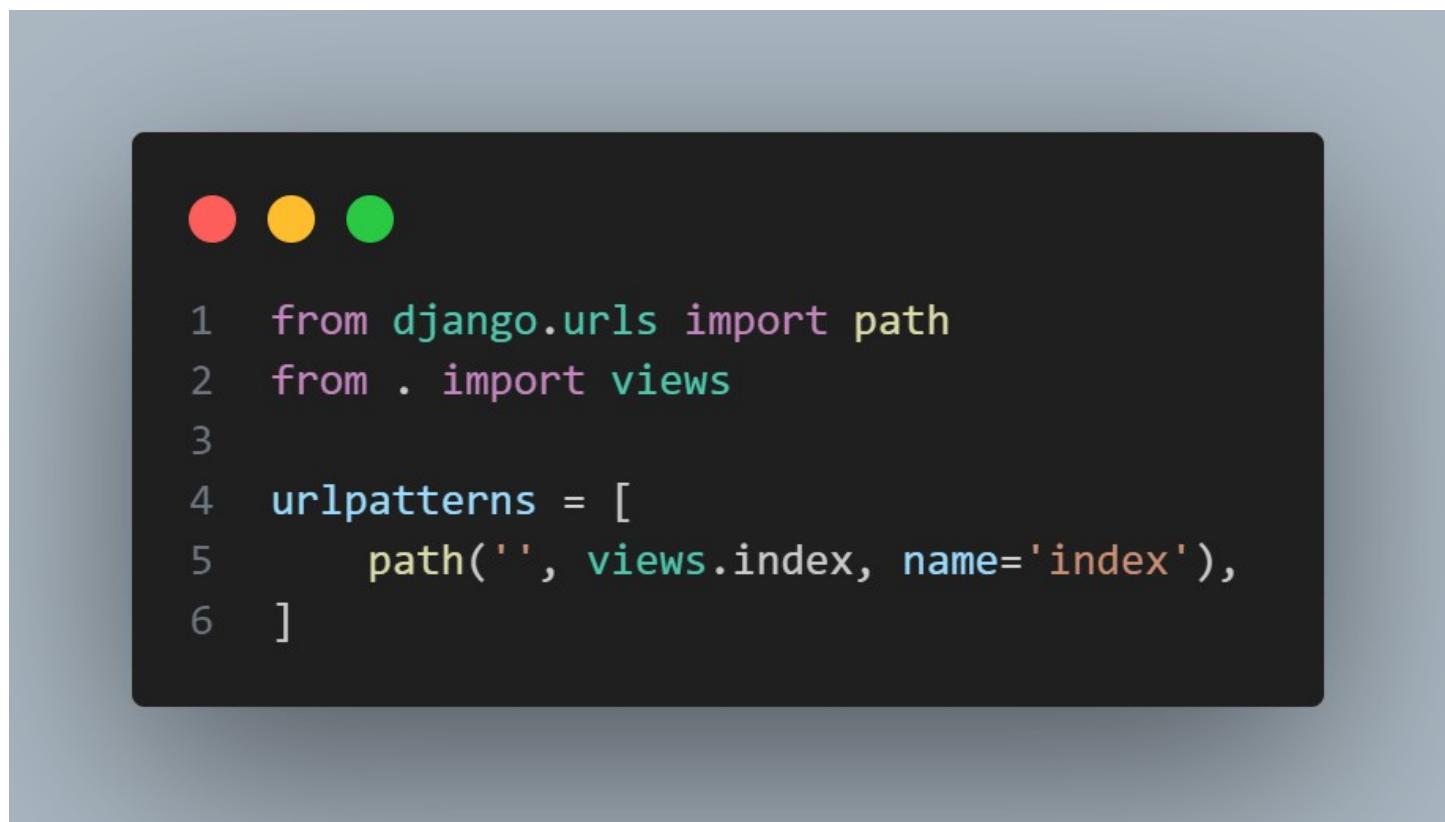
В Django для создания ссылок часто используются шаблоны и функция url. Пример использования:



```
1 <a href="{% url 'index' %}>Главная страница</a>
```

Рисунок 86 - Функция url в шаблонизаторе

В данном случае 'index' — это имя маршрута, определенного в файле urls.py.



```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('', views.index, name='index'),
6 ]
```

Рисунок 87 - Файл urls.py

Медиафайлы в Django

Медиафайлы включают в себя изображения, видео, аудио и другие файлы, которые могут быть загружены и использованы на веб-сайте. В Django работа с медиафайлами организована следующим образом:

1. Настройка конфигурации проекта для работы с медиафайлами:

В файле settings.py необходимо указать пути для медиафайлов:

```
1 MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
2 MEDIA_URL = '/media/'
```

Рисунок 88 - Создание пути для сохранения файлов

Настройка маршрутов для обслуживания медиафайлов:

В файле urls.py необходимо добавить маршруты для медиафайлов при разработке:

```
1 from django.conf import settings
2 from django.conf.urls.static import static
3
4 urlpatterns = [
5     # другие маршруты
6 ]
7
8 if settings.DEBUG:
9     urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Рисунок 89 - Основной файл urls.py

Модель с полем для загрузки медиафайлов:

В Django существует специальное поле FileField и ImageField для работы с файлами и изображениями соответственно. Пример модели:



```
1 from django.db import models
2
3 class MyModel(models.Model):
4     title = models.CharField(max_length=100)
5     image = models.ImageField(upload_to='images/')
```

Рисунок 90 - Классы для добавления

Форма для загрузки файлов:

Для создания формы загрузки файлов используется ModelForm:



```
1 from django import forms
2 from .models import MyModel
3
4 class MyModelForm(forms.ModelForm):
5     class Meta:
6         model = MyModel
7         fields = ['title', 'image']
```

Рисунок 91 - Формы для модели

Шаблон для отображения и загрузки медиафайлов:

Пример шаблона, который позволяет загружать и отображать изображение:

```
1  <form method="post" enctype="multipart/form-data">
2      {% csrf_token %}
3      {{ form.as_p }}
4      <button type="submit">Загрузить</button>
5  </form>
6
7  {% if object.image %}
```

Рисунок 92 - Создание формы

ПРАКТИЧЕСКАЯ ЧАСТЬ

Всю практическую выполняйте в своем проекте из прошлого задания

Медиафайл

1. Заходим в settings.py. Создаем две переменные MEDIA_ROOT и MEDIA_URL.



```
1 MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
2 MEDIA_URL = '/media/'
```

Рисунок 93 - Файл settings.py

С помощью этих двух переменных, django будет определять куда сохранять медиафайлы.

2. Теперь заходит в основной urls.py, и пишем данное условие



```
1 from django.contrib import admin
2 from django.urls import path, include
3 from django.conf.urls.static import static
4 from django.conf import settings
5
6
7 urlpatterns = [
8     path('admin/', admin.site.urls),
9 ]
10
11
12
13 if settings.DEBUG:
14     urlpatterns += static(settings.MEDIA_URL, document_root = settings.MEDIA_ROOT)
```

Рисунок 94 - Основной urls.py

Это условия нужно чтобы локальные сервер обрабатывал медиафайлы. (в продакшене используют Nginx или Apache)

3. Добавляем в модель поля image



```
1  image = models.ImageField(upload_to="book/%Y/%m/%d")
```

Рисунок 95 - Использование ImageField

В данном поле заполняем аргумент upload_to, данный аргумент определяет как будет сохраняться файл в папке media.

После написание данного кода будет выводиться ошибка(если у вас включен сервер)

```
ERRORS:  
book.Book.image: (fields.E210) Cannot use ImageField because Pillow is not installed.  
      HINT: Get Pillow at https://pypi.org/project/Pillow/ or run command "python -m pip install Pillow".
```

Рисунок 96 - Ошибка Pillow

В данной ошибки написано, чтобы использовать поле ImageField нужно установить библиотеку Pillow

4. Заходим в админ-панель и добавляем изображение
5. Создаем функцию для отображение страницы



```
1  def get_books(request):  
2      books = Book.objects.all()  
3      return render(request, 'home.html', context= {  
4          'books' : books  
5      })
```

Рисунок 97 - Функция get_books

(Сделайте по своей тематике веб-приложения)

6. Пишем в html файл данный код



Рисунок 98 - Отображение изображений с помощью шаблонизатора

7. Теперь на странице будет отображаться изображение



Рисунок 99 - Веб-страница

Ссылки

1. Для создания динамически ссылок откройте в вашем приложении файл urls.py



```
1 from django.urls import path
2 from .views import get_books
3
4 urlpatterns = [
5     path('', get_books, name='home'),
6 ]
```

Рисунок 100 - Файл urls.py

Третьем аргументам напишите name и присвойте какой-либо название.

(у вас будет по другому)

2. Заходим в html файл. В тег <a> пишем {% url 'home' %}



```
1 <a href="{% url 'home' %}">Главная страница</a>
```

Теперь в теги <a> , есть динамическая ссылка.

ЗАДАНИЕ

1. Добавьте ссылки в свой проект
2. Сделайте страницу для отображений всех товаров

Контрольные вопросы

1. Для чего нужен аргумент name в функций path?
2. Для чего нужная библиотеку pillow?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №9

Динамические ссылки

ЦЕЛЬ ЗАНЯТИЯ

Создание страниц книги

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Что такое динамические ссылки?

Динамические ссылки — это URL, которые включают переменные части, что позволяет создавать адреса, зависящие от данных. Например, если у вас есть блог, вы можете использовать динамические ссылки для создания URL-адресов для каждого поста на основе его идентификатора или слага (например, /posts/1/ или /posts/my-first-post/).

Настройка динамических ссылок в Django

Для настройки динамических ссылок в Django нужно работать с несколькими компонентами фреймворка:

1. URLs (`urls.py`)
2. Views (`views.py`)
3. Шаблоны (`templates`)

URLs

Файл `urls.py` отвечает за маршрутизацию запросов к соответствующим представлениям (`views`). Чтобы создать динамическую ссылку, нужно определить маршрут с переменной частью.

Пример:

```
● ● ●
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('posts/<int:id>', views.post_detail, name='post_detail'),
6     path('posts/<slug:slug>', views.post_detail_by_slug, name='post_detail_by_slug'),
7 ]
```

Рисунок 101 - Файл `urls.py`

Здесь `<int:id>` и `<slug:slug>` — это динамические части URL, которые будут переданы в соответствующее представление как параметры.

Views

Представления (`views`) обрабатывают запросы и возвращают ответы. Они получают параметры из URL и используют их для выполнения логики.

Пример:



```
1 from django.shortcuts import render, get_object_or_404
2 from .models import Post
3
4 def post_detail(request, id):
5     post = get_object_or_404(Post, id=id)
6     return render(request, 'post_detail.html', {'post': post})
7
8 def post_detail_by_slug(request, slug):
9     post = get_object_or_404(Post, slug=slug)
10    return render(request, 'post_detail.html', {'post': post})
11
```

Рисунок 102 - Файл views.py

В этом примере функции post_detail и post_detail_by_slug принимают параметры id и slug соответственно, чтобы получить конкретный пост из базы данных и отобразить его.

Шаблоны

Шаблоны используют для отображения данных. В них можно создавать ссылки на другие страницы, включая динамические ссылки.

Пример:



```
1 <a href="{% url 'post_detail' id=post.id %}">{{ post.title }}</a>
```

Рисунок 103 - Динамическая ссылка

Этот тег `{% url 'post_detail' id=post.id %}` генерирует URL, используя имя маршрута post_detail и значение переменной post.id.

Обратное разрешение URL

Django предоставляет мощный инструмент для создания динамических ссылок — функцию reverse, которая позволяет генерировать URL на основе имен маршрутов и параметров. Это особенно полезно, когда вам нужно программно создавать ссылки.

Пример:



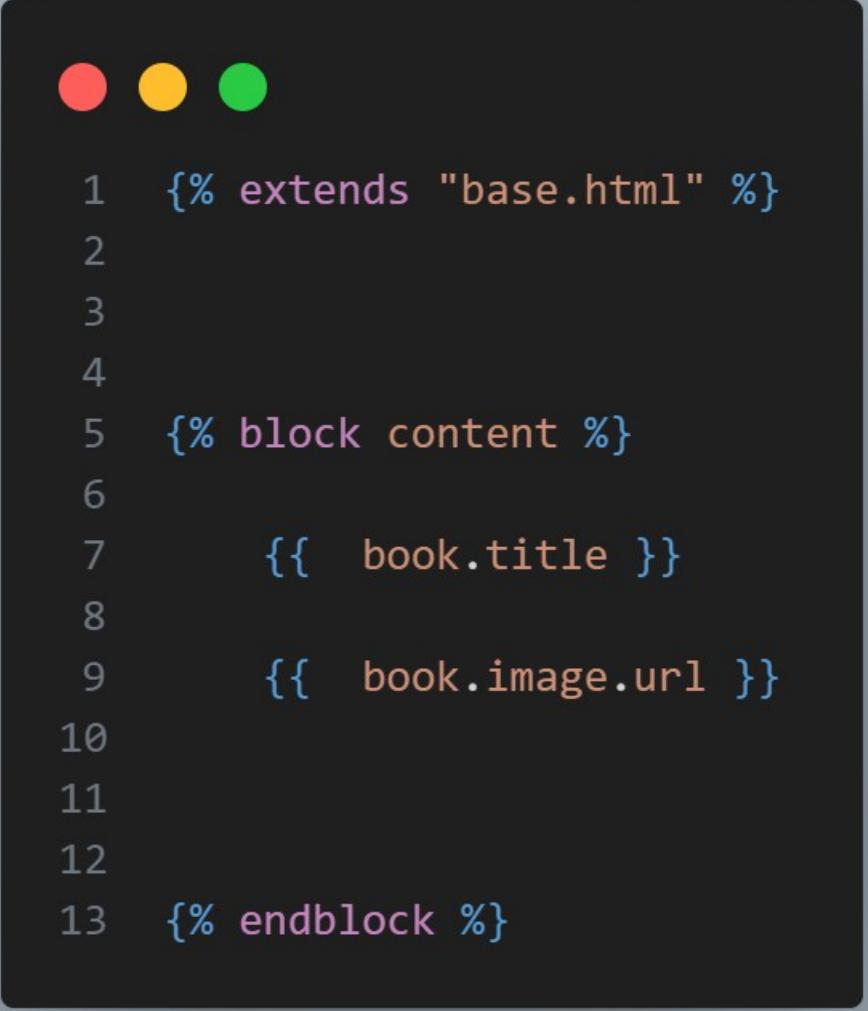
```
1 from django.urls import reverse
2
3 def get_post_url(post):
4     return reverse('post_detail', args=[post.id])
```

Рисунок 104 - Использование функции reverse

Функция reverse генерирует URL, используя имя маршрута post_detail и значение параметра post.id.

ПРАКТИЧЕСКАЯ ЧАСТЬ

1. Создаем папке templates, файл detail_book.html



```
1  {% extends "base.html" %}  
2  
3  
4  
5  {% block content %}  
6  
7      {{ book.title }}  
8  
9      {{ book.image.url }}  
10  
11  
12  
13  {% endblock %}
```

Рисунок 105 - Файл detail_book.html

2. Заходим в views.py. Создаем функцию get_book_detail, и добавляем вторым аргументам pk

```
1
2 def get_book_detail(request, pk):
3     book = Book.objects.get(pk=pk)
4     return render(request, 'detail_book.html', context={
5         'book':book
6     })
```

Рисунок 106 - Функция для отображения продукта

3. В urls.py добавляем путь

```
1     path('book/<int:pk>', get_book_detail, name='book_detail')
2
```

Рисунок 107 - Путь по pk

4. Заходим в html файл в котором отображаются все товары, и добавляем данный код в цикл

```
1 <a href="{% url 'book_detail' i.id %}">{{i.title}}</a>
2
```

Рисунок 108 - Ссылка на страницу продукта

5. После перехода по ссылки будет открываться страница продукта.

- [Главная страница](#)
- [О нас](#)



Евгений Онегин

ЗАДАНИЕ

1. Сделайте персональные страницы для своего продукта по тематике веб-приложения

Контрольные вопросы

1. Для чего нужный персональные ссылки?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №10

Http запросы

ЦЕЛЬ ЗАНЯТИЯ

Отправка http запросов

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Формы — это важная часть веб-приложений, позволяющая пользователям вводить и отправлять данные на сервер. В Django формы используются для ввода данных, их проверки и обработки. Отправка HTTP-запросов (GET и POST) является основным способом взаимодействия клиента и сервера в веб-приложениях.

Формы в Django

Django предоставляет мощный механизм для работы с формами через модуль `django.forms`. Формы в Django обеспечивают следующие функции:

- Валидация данных.
- Генерация HTML-форм.
- Обработка и сохранение данных.

Создание форм

В Django формы создаются с использованием классов, которые наследуются от `django.forms.Form` или `django.forms.ModelForm`.

Пример формы на основе Form:

```
1 from django import forms
2
3 class ContactForm(forms.Form):
4     name = forms.CharField(label='Your name', max_length=100)
5     email = forms.EmailField(label='Your email')
6     message = forms.CharField(widget=forms.Textarea, label='Your message')
```

Рисунок 109 - Создание формы

Валидация форм

Django автоматически выполняет валидацию данных, введенных в форму. Для дополнительной проверки можно переопределить метод `clean` или методы `clean_<fieldname>`.



```
1 class ContactForm(forms.Form):
2     ...
3     def clean_email(self):
4         email = self.cleaned_data['email']
5         if not email.endswith('@example.com'):
6             raise forms.ValidationError("Email must be from example.com domain")
7         return email
```

Рисунок 110 - Создание валидации в форме

Обработка HTTP-запросов

HTTP-запросы в Django обрабатываются с использованием представлений (views). Существует два основных типа HTTP-запросов, используемых для работы с формами: GET и POST.

- **GET-запросы:** используются для получения данных с сервера. Форма отображается пользователю.
- **POST-запросы:** используются для отправки данных на сервер. Форма отправляется на сервер для обработки.

Пример представления для обработки формы



```
1 from django.shortcuts import render
2 from django.http import HttpResponseRedirect
3 from .forms import ContactForm
4
5 def contact_view(request):
6     if request.method == 'POST':
7         form = ContactForm(request.POST)
8         if form.is_valid():
9             # Обработка данных формы
10            name = form.cleaned_data['name']
11            email = form.cleaned_data['email']
12            message = form.cleaned_data['message']
13            # Можно добавить логику для сохранения данных или отправки email
14            return HttpResponseRedirect('/thanks/')
15     else:
16         form = ContactForm()
17
18     return render(request, 'contact.html', {'form': form})
```

Рисунок 111 - Файл views.py. Обработка http запросов

Шаблоны для отображения форм

Шаблоны используются для отображения HTML-форм. Django предоставляет специальный синтаксис для рендеринга форм.

Пример шаблона:



```
1 <form method="post">
2     {% csrf_token %}
3     {{ form.as_p }}
4     <button type="submit">Send</button>
5 </form>
```

Рисунок 112 - Форма в html шаблоне

Здесь `{% csrf_token %}` используется для защиты от CSRF-атак (Cross-Site Request Forgery).

CSRF-токен (Cross-Site Request Forgery Token)

CSRF-токен — это случайное значение, создаваемое сервером и включаемое в каждую HTML-форму, которая отправляется с использованием метода POST. Этот токен затем проверяется сервером на соответствие при получении POST-запроса, что помогает предотвратить атаки типа CSRF.

Зачем нужен CSRF-токен?

CSRF-атаки позволяют злоумышленникам выполнять действия от имени пользователя без его ведома. Например, если пользователь вошел в свою учетную запись на сайте, злоумышленник может заставить его браузер выполнить запрос на изменение пароля, если CSRF-токен не используется.

Как работает CSRF-токен?

- Генерация токена:** Когда сервер отправляет форму клиенту, он генерирует уникальный CSRF-токен и включает его в форму.
- Отправка формы:** Когда пользователь отправляет форму, CSRF-токен включается в POST-запрос.
- Проверка токена:** Сервер проверяет, что полученный CSRF-токен совпадает с тем, который был сгенерирован для данной сессии.

Обработка результатов формы

После отправки формы данные должны быть обработаны на сервере. Обычно это включает:

1. Проверку данных (валидацию).
2. Обработку данных (сохранение в базу данных, отправка email и т.д.).
3. Перенаправление пользователя на другую страницу или отображение сообщения об успехе.

Пример обработки данных:



```
1 if form.is_valid():
2     # Сохранение данных в базу
3     new_post = form.save(commit=False)
4     new_post.author = request.user
5     new_post.save()
6     return HttpResponseRedirect('/posts/')
```

Рисунок 113 - Обработка данных

Формы и отправка HTTP-запросов — это основные механизмы взаимодействия пользователя с веб-приложением в Django. Правильное использование форм, валидации данных и обработки HTTP-запросов позволяет создавать надежные и безопасные веб-приложения. Django предоставляет богатый набор инструментов для работы с формами, что облегчает разработку и поддержание веб-приложений.

ПРАКТИЧЕСКАЯ ЧАСТЬ

В этой практической части, сделаем поиск, чтобы научиться отправлять http запросы на сервер. В django есть два способа:

1. Через формы в html
2. Через класс формы в django

Для этой практической преподаватель выдаст проект за девять практических. Посмотри этот проект, и добавьте достающие части кода, а также проанализируйте проекта.

1. В файле views.py, функцию search_book

```
1  from django.db.models import Q
2
3  def search_book(request):
4      if request.method == "GET":
5          search = request.GET['search']
6          books = Book.objects.filter(
7              Q(title__icontains = search) | Q(author__name__icontains=search))
8          return render(request, template_name='books.html', context={
9              'books' : books,
10             'title' : 'Книги'
11         })
12     return redirect(reverse('home'))
```

Рисунок 114 - Функция поиска

В данной функцией, есть условия с помощью, которой определяется тип запроса(request). После определение типа запроса, извлекаем данные из запроса под ключем search. Для поиска используем сложные запросы. Импортируем класс Q, чтобы создать сложный. В запросе фильтруем книги по названию книги без учета регистра(icontains), и по автору без учета регистра(icontains). Передаем данные в шаблон.(не забудьте создать url)

2. В header-е создаем форму.

```
1  <div class="search">
2      <form action="{% url 'search' %}">
3          <input type="text" name="search">
4          <button type="submit"> Поиск</button>
5      </form>
6  </div>
```

Рисунок 115 - Форма для поиска книг

Форме в input должен быть name, с помощью name определяем информацию. Это можно посмотреть на Рисунке 114 на пятой строчке.

3. Пишем например букву «M»



Рисунок 116 - Выдача поиска

ЗАДАНИЕ

- Добавьте поиск в свой проект

Контрольные вопросы

Информацию искать в самому

- Какие http запросы существуют?
- Какие запросы может отдавать html файл?
- Что чего нужен каждый запрос?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №11

Формы django и регистрация

ЦЕЛЬ ЗАНЯТИЯ

1. Создание форм
2. Регистрация пользователя

ПРАКТИЧЕСКАЯ ЧАСТЬ

В этой практической будем использовать формы django для отправки запросов. У django есть встроенные модели для пользователей, а также формы для регистрации и авторизации.

1. Создайте новое приложение, например account (это мы делаем, чтобы разбить логику в проекте)
2. В приложение создаем файл forms.py. В файл импортируем forms и форму для регистрации.

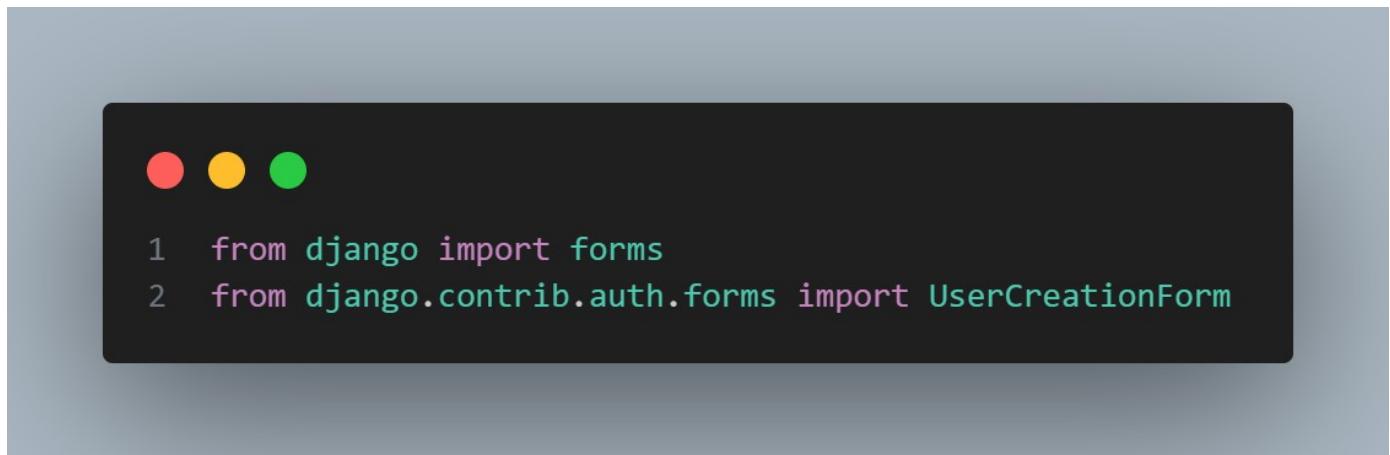


Рисунок 117 - Импорт форм для регистрации

От UserCreationForm будем наследовать для создания персональной формы для регистрации в нашем проекте.

3. Создаем класс RegisterForm. Наследуемся от класса UserCreationForm. В классе RegisterForm добавляем атрибуты, которые будут отображаться в форме в html. В атрибуты добавляем типы данных, через forms, например forms.CharField.

```
● ● ●
```

```
1 from django import forms
2 from django.contrib.auth.forms import AuthenticationForm, UserCreationForm
3 from django.core.validators import RegexValidator
4
5
6 class RegisterForm(UserCreationForm):
7     username = forms.CharField(
8         widget=forms.TextInput(
9             attrs={
10                 'autocomplete': 'text',
11                 'placeholder': 'Введите логин',
12                 }
13             ),
14             required=False,
15             validators=[RegexValidator(r'^[0-9а-яА-ЯёЁ]', "Введите логин латиницой")],
16         )
17     email = forms.EmailField(
18         widget=forms.EmailInput(
19             attrs={
20                 'autocomplete': 'email',
21                 'placeholder': 'Введите электрону почту ',
22                 }
23             ),
24             required=False
25         )
26     password1 = forms.CharField(
27         widget=forms.PasswordInput(
28             attrs={
29                 'placeholder': 'Введите пароль ',
30                 }
31             ),
32             required=False
33         )
34     password2 = forms.CharField(
35         widget=forms.PasswordInput(
36             attrs={
37                 'placeholder': 'Повторите пароль',
38                 }
39             ),
40             required=False
41         )
```

Рисунок 118 - Форма с атрибутами класса

4. В файле views.py создаем функцию для регистрации.



```
1  from django.shortcuts import render, redirect
2  from django.contrib.auth import login, authenticate
3  from .forms import RegisterForm
4  from django.http import HttpRequest
5
6
7
8
9  def register(request: HttpRequest):
10     if request.method == 'POST':
11         form = RegisterForm(request.POST)
12         if form.is_valid():
13             user = form.save()
14             user.save()
15             return redirect('home')
16     else:
17         form = RegisterForm()
18     return render(request, 'register.html', context={
19         'title': 'Регистрация',
20         'form': form,
21     })
```

Рисунок 119 - Функция регистрации

В функции создаем логику для регистрации. Создаем условие, чтобы принимать POST запросы.

Используем класс форму, чтобы сериализировать данные из запроса. Создаем еще одной условия для проверки валидности данных, а после сохраняем.

5. Создаем html файл для формы.



```
1  {% extends "base.html" %}  
2  
3  
4  {% block content %}  
5  
6      <form method="POST">  
7          {% csrf_token %}  
8          {{ form.as_p }}  
9  
10         <button type="submit"> Зарегистрироваться </button>  
11     </form>  
12  
13  {% endblock %}
```

Рисунок 120 - Html шаблон

6. Теперь на страницы будем показываться форма



Username: [введите логин]
Password1: [введите пароль]
Password2: [повторите пароль]
Email: [введите электронную почту]
Зарегистрироваться

Рисунок 121 - Страница регистрации

Если отправить форму пустой, то пользователь зарегистрируется, потому что в классе нету валидации.

7. Пишем в классе RegisterForm



```
1 def clean_password1(self):
2     password = self.cleaned_data['password1']
3     if password == '':
4         raise forms.ValidationError('Введите пароль', code='invalid')
5     return password
6
7 def clean_username(self):
8     username = self.cleaned_data['username']
9     if username == '':
10        raise forms.ValidationError('Введите логин', code='invalid')
11    return username
12
13 def clean_email(self):
14     email = self.cleaned_data['email']
15     if email == '':
16        raise forms.ValidationError('Введите электронную почту', code='invalid')
17    return email
18
19
20 class Meta(UserCreationForm.Meta):
21     fields = ("username", "email", "password1", "password2")
```

Рисунок 122 - Валидация полей в форме

Теперь при регистрации пользователя будем проверка на пустое поле

ЗАДАНИЕ

1. Добавьте регистрацию в свой проект

Контрольные вопросы

1. Что такое CSRF-токен?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №12

Авторизация

ЦЕЛЬ ЗАНЯТИЯ

1. Авторизация пользователя

ПРАКТИЧЕСКАЯ ЧАСТЬ

1. В файле forms.py создаем форму для авторизации



```
1  class LoginForm(AuthenticationForm):
2      username = forms.CharField(
3          max_length=254,
4          widget=forms.TextInput(
5              attrs={
6                  'autocomplete': 'text',
7                  'placeholder': 'Логин',
8              }
9          ),
10         required=False
11     )
12     password = forms.CharField(
13         widget=forms.PasswordInput(
14             attrs={
15                 "autocomplete": "current-password",
16                 'placeholder': 'Пароль',
17             }
18         ),
19         required=False
20     )
21
22     error_messages = {
23         "invalid_login": (
24             "Введите логин и пароль правильно"
25         ),
26     }
27
28
29     def clean_password(self):
30         password = self.cleaned_data['password']
31         if password == '':
32             raise forms.ValidationError('Введите пароль', code='invalid')
33         return password
34     def clean_username(self):
35         username = self.cleaned_data['username']
36         if username == '':
37             raise forms.ValidationError('Введите логин ', code='invalid')
38         if not User.objects.filter(username=username):
39             raise forms.ValidationError('Такого пользователя не существует', code='invalid')
40         return username
```

Рисунок 123 - Форма для авторизации пользователя с валидацией

2. Заходим в файл views.py и пишем данный код.



```
1 def login_user(request: HttpRequest):
2     if request.method == 'POST':
3         form = LoginForm(request, data=request.POST)
4         if form.is_valid():
5             user = form.get_user()
6             if user is not None:
7                 login(request, user)
8                 return redirect('home')
9     else:
10         form = LoginForm()
11     return render(request, 'login.html', context={
12         'title': 'Авторизация',
13         'form' : form,
14
15     })
```

Рисунок 124 - Функция для авторизации

Также создайте html шаблон и url.

3. При авторизации пользователя вас перекинет на главную страницу, чтобы убедиться вы авторизовались добавим код в шапку сайта



```
1  {% if request.user %}  
2      {{ request.user.username }}  
3  {% endif %}
```

Рисунок 125 - Отображение авторизованного пользователя

Теперь при авторизации будет показываться username пользователя

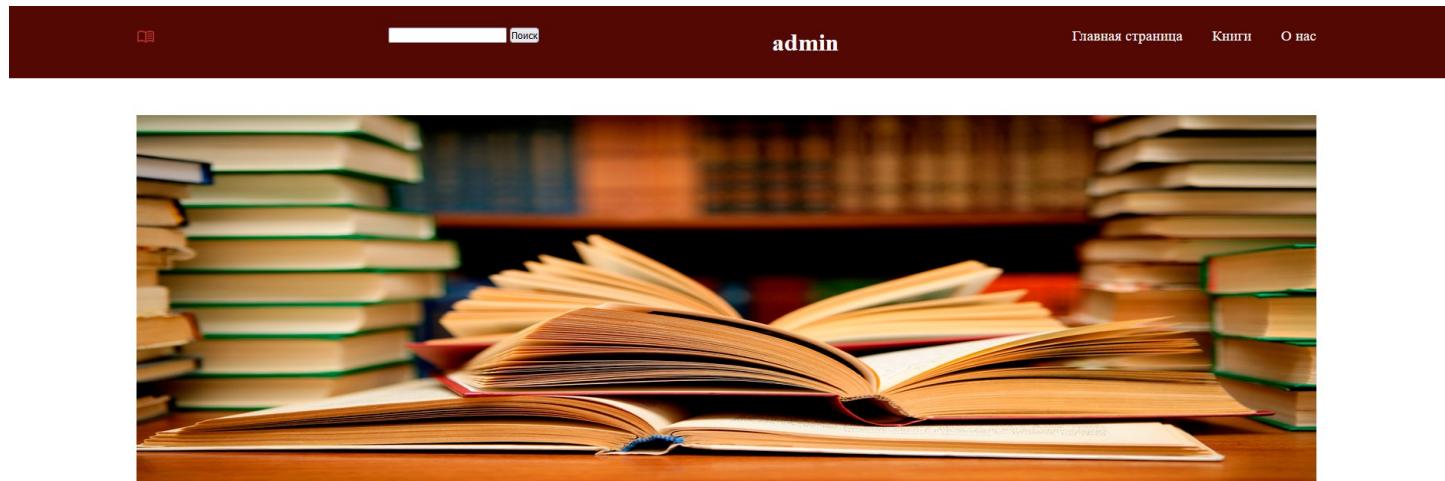


Рисунок 125 - Главная страница пользователя

ЗАДАНИЕ

1. Добавьте авторизацию в свой проект
2. Добавьте выход в свой проект(используйте функцию logout)

Контрольные вопросы

1. Чем отличает авторизация от аутентификации?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №13

Аутентификации

ЦЕЛЬ ЗАНЯТИЯ

1. Узнать что такое аутентификации
2. Создания профиля

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Аутентификация — это процесс проверки подлинности пользователя, т.е. подтверждение, что пользователь является тем, за кого он себя выдает. Django поддерживает различные методы аутентификации, что позволяет разработчикам выбирать наиболее подходящий для их нужд.

Основные типы аутентификации

1. Основная аутентификация (Basic Authentication)

Основная аутентификация требует от пользователя предоставления имени пользователя и пароля, которые отправляются на сервер в заголовках HTTP-запроса.

Преимущества:

- Простота реализации.
- Поддержка большинством веб-браузеров и библиотек HTTP-клиентов.

Недостатки:

- Пароли передаются в открытом виде (без шифрования), что требует использования HTTPS.
- Ограниченные возможности управления сессиями.

2. Токен-аутентификация (Token Authentication)

Токен-аутентификация использует уникальные токены для идентификации пользователей. Пользователь получает токен после успешной аутентификации, который затем используется для всех последующих запросов.

Преимущества:

- Безопасность передачи данных.
- Гибкость и масштабируемость (токены могут быть легко отозваны или обновлены).

Недостатки:

- Необходимость управления токенами (хранение, выдача, отзыв).
- Более сложная реализация по сравнению с базовой аутентификацией.

3. JSON Web Token (JWT) аутентификация

JWT — это стандарт для создания токенов, содержащих JSON-данные. Эти токены могут быть подписаны и зашифрованы для обеспечения безопасности.

Преимущества:

- Независимость от сервера (токены могут быть проверены на любом сервере).
- Возможность хранения дополнительной информации в токене.

Недостатки:

- Токены могут быть подвержены атакам при неправильной настройке.
- Необходимость управления токенами.

4. OAuth2 аутентификация

OAuth2 — это стандарт протокола авторизации, который позволяет третьим сторонам получать ограниченный доступ к ресурсам пользователя без передачи пароля.

Преимущества:

- Безопасность (использует токены доступа вместо паролей).
- Широкая поддержка различными сервисами (Google, Facebook, GitHub и т.д.).

Недостатки:

- Сложность настройки и реализации.
- Зависимость от сторонних сервисов.

5. Аутентификация на основе сессий (Session Authentication)

Django по умолчанию использует сессионную аутентификацию, которая сохраняет информацию о пользователе в сессии после успешного входа.

Преимущества:

- Простой механизм для веб-приложений.
- Автоматическое управление сессиями.

Недостатки:

- Не подходит для API, если требуется stateless аутентификация.
- Требует использования cookies, что может быть ограничением для некоторых клиентских приложений.

ПРАКТИЧЕСКАЯ ЧАСТЬ

В этой практической создадим профиль для пользователя. Для начала надо определим как это сделаем, есть три способа:

1. Расширить изначальной модели пользователя(AbstractUser) в django
2. Создать модель профиль со связью с моделью пользователя
3. Использование встроенной модели пользователя с профилем в одной модели через наследование AbstractBaseUser

Из перечисленных вариантов, лучше всего использовать 2 способ для нашего.

При использование второго варианта:

1. Разделение логики в пользователя и профиля
2. Легкое добавление и изменение полей в профиле без влияние на основную модель пользователям

1. В приложение account создаем модель profile. В модели создаем поле user, в котором будем использовать связь один ко одному. После этих действий, создаем другие поля для профиля с атрибутом blank=True. Это атрибут говорит, что поле не обязательно для заполнения.

```
1  from django.db import models
2  from django.contrib.auth.models import User
3
4
5  class Gender(models.TextChoices):
6      MEN = "Мужской"
7      WOMEN = "Женский"
8
9
10 class Profile(models.Model):
11     user = models.OneToOneField(User, on_delete=models.CASCADE)
12     gender = models.CharField(choices=Gender, blank=True, max_length=20)
13
14     country = models.CharField(max_length=100, blank=True)
15     city = models.CharField(max_length=100, blank=True)
16     street = models.CharField(max_length=100, blank=True)
17     house = models.CharField(max_length=100, blank=True)
18     apartment_number = models.CharField(max_length=100, blank=True)
```

Рисунок 126 - Модель пользователя

2. Теперь нужно сделать так, чтобы при регистрации пользователя, создавался профиль. Это логику можно написать в функцией register в файле views, но в django, есть «диспетчер сигналов» .

Диспетчер сигналов помогает разделенным приложениям получать уведомления о действиях, происходящих в других частях проекта.

Создаем файл signals.py в приложении account. В файле импортируем декоратор receiver и post_save.



```
1 from django.db.models.signals import post_save
2 from django.dispatch import receiver
3 from django.contrib.auth.models import User
4 from .models import Profile
5
6 @receiver(post_save, sender=User)
7 def create_user_profile(sender, instance, created, **kwargs):
8     if created:
9         Profile.objects.create(user=instance)
10
11 @receiver(post_save, sender=User)
12 def save_user_profile(sender, instance, **kwargs):
13     instance.profile.save()
```

Рисунок 127 - Файл signals.py

Создаем функцию, которая будет создавать профиль. Перед функцией используем декоратор receiver, чтобы при регистрации пользователя данная функция активировалась.

В декораторе receiver, первом аргумент отдаляем тип сигналов(или лист сигналов), после первого аргумента создаем переменную, которую будем передавать в функцию(переменная в декораторе и функций должны быть одинаковыми). В аргументе instance храниться экземпляр пользователя.

Также функцией, есть аргумент created из за сигнала post_save.

3. В приложение account, заходим файл apps.py, и в классе конфига приложения изменяем метод класса ready.



```
1 from django.apps import AppConfig  
2  
3  
4 class AccountConfig(AppConfig):  
5     default_auto_field = 'django.db.models.BigAutoField'  
6     name = 'account'  
7  
8  
9     def ready(self) -> None:  
10         import account.signals
```

Рисунок 128 - Файл apps.py в приложении account

В методе ready импортируем файл с сигналами, чтобы их инициализировать.

Теперь после регистрации будет создаваться таблица профиля пользователя.

4. Создаем функцию для отображение профиля. Для этой функции используем декоратор login_required для аутентификации пользователя, если пользователя не аутентифицированный, то его перекидывают на страницу авторизации



```
1 from django.contrib.auth.decorators import login_required  
2  
3  
4 @login_required(login_url='login')  
5 def profile_view(request: HttpRequest):  
6     user = Profile.objects.select_related('user').get(user=request.user)  
7     return render(request, 'profile.html', context={  
8         'user' : user  
9     })
```

Рисунок 129 - Функция для отображения профиля

В переменной user создаем запрос, которые объединяет две таблицы (аналог join из БД)

5.



```
1  {% extends "base.html" %}  
2  
3  
4  {% block content %}  
5  
6  
7      <h2> Пользователь:{{ user.user.username }}</h2>  
8  
9      <h2> Пол: {{ user.gender}}</h2>  
10  
11     <h2> Город: {{ user.city }}</h2>  
12  
13  {% endblock %}
```

Рисунок 130 - Профиль пользователя

Чтобы отобразить данные из модели пользователя нужно два раза ссылаться на переменную user, потому что мы ссылаемся на связь user.

ЗАДАНИЕ

1. Создайте форму для изменения данных профиля (удаление, изменения, создание)

Контрольные вопросы

1. Какой тип аутентификации используется в django?
2. Для чего нужна аутентификация?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №14

Кастомные сессии и context_processors

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Сессии в Django позволяют хранить данные для конкретных пользователей между запросами. По умолчанию, Django использует файлы cookies для идентификации сессий и хранит данные в базе данных. Однако, вы можете настроить кастомные сессии, используя различные бекенды (backends) или настраивая собственные механизмы хранения.

Основные моменты:

1. **Настройка типа бекенда сессий:** Django поддерживает несколько бекендов для хранения сессий:

- База данных (`django.contrib.sessions.backends.db`)
- Куки (`django.contrib.sessions.backends.signed_cookies`)
- Кэш (`django.contrib.sessions.backends.cache`)
- Файловая система (`django.contrib.sessions.backends.file`)
- Кэш+база данных (`django.contrib.sessions.backends.cached_db`)

Выбор бекенда определяется в настройках проекта (`settings.py`):



Рисунок 131 - Изменение сессий

Кастомизация данных сессий: Для добавления кастомной логики в работу сессий, можно создать собственные middleware или subclass (наследовать) от существующего бекенда.

```
1 from django.contrib.sessions.backends.db import SessionStore as DBStore
2
3 class CustomSessionStore(DBStore):
4     def custom_method(self):
5         # Ваша логика
6         pass
```

Рисунок 132 - Кастомная сессия

И затем указать кастомный класс в настройках:

```
1 SESSION_ENGINE = 'path.to.CustomSessionStore'
```

Рисунок 133 - Добавление кастомной сессий

Настройки и параметры сессий: В `settings.py` можно указать дополнительные параметры для управления сессиями:

- `SESSION_COOKIE_NAME`: Имя куки, по умолчанию `sessionid`.
- `SESSION_COOKIE_AGE`: Время жизни куки, по умолчанию 1209600 секунд (2 недели).
- `SESSION_SAVE_EVERY_REQUEST`: Если `True`, то сессия будет сохраняться после каждого запроса.

Context processors – это функции, которые добавляют переменные в контекст шаблонов глобально. Они позволяют сделать определенные данные доступными для всех шаблонов без необходимости передавать их в каждом `view`.

Основные моменты:

1. **Создание контекстного процессора:** Контекстный процессор — это функция, которая принимает объект запроса (`request`) и возвращает словарь данных.



```
1 def custom_context_processor(request):
2     return {
3         'custom_variable': 'Hello, World!'
4     }
5
```

Рисунок 134 - Функция контекст процесса

Регистрация контекстного процессора: Контекстные процессоры регистрируются в настройках проекта (`settings.py`) в разделе `TEMPLATES`:



```
1 TEMPLATES = [
2     {
3         'BACKEND': 'django.template.backends.django.DjangoTemplates',
4         'DIRS': [],
5         'APP_DIRS': True,
6         'OPTIONS': {
7             'context_processors': [
8                 'django.template.context_processors.debug',
9                 'django.template.context_processors.request',
10                # Ваш кастомный процессор
11                'path.to.custom_context_processor',
12            ],
13        },
14    },
15]
```

Рисунок 135 - Регистрация контекст процессов в шаблонах

Использование контекстных процессоров: Переменные, добавленные контекстным процессором, будут доступны во всех шаблонах без необходимости явно передавать их из `view`.



```
1      <!-- В вашем шаблоне -->
2  <p>{{ custom_variable }}</p>
```

Рисунок 136 - Использование переменной в шаблоне

ПРАКТИЧЕСКАЯ ЧАСТЬ

(Добавьте модель вашего продукта цену)

Создаем приложение cart, чтобы разделить логику приложения. Также создаем cart.py в которой будет логика корзины

1. В файле cart.py создаем класс CartSession, который будет наследоваться от класса SessionBase.

Создаем атрибут для класса под названием CART_SESSION_ID со значением «cart», затем создаем атрибуты экземпляра класса session_key и cart

```
● ● ●  
1  from django.contrib.sessions.backends.base import SessionBase  
2  from book.models import Book  
3  
4  class CartSession(SessionBase):  
5      CART_SESSION_ID = 'cart'  
6  
7  
8      def __init__(self, session: dict) -> None:  
9          self.session : dict = session  
10         self.cart = self.session.get(self.CART_SESSION_ID)  
11  
12         if not self.cart:  
13  
14             self.cart = self.session[self.CART_SESSION_ID] = {}
```

Рисунок 137 - Класс CartSession с конструктором

2. В классе добавляем магический метод __iter__ в этот метод нужен, чтобы создать свою итерацию



```
1 def __iter__(self):
2
3     book_ids = self.cart.keys()
4
5     books = Book.objects.filter(id__in=book_ids)
6
7     cart = self.cart.copy()
8
9     for book in books:
10         cart[str(book.id)]['book'] = book
11
12     for item in cart.values():
13         item['price'] = int(item['price'])
14         item['total_price'] = item['price'] * item['quantity']
15         yield item
```

Рисунок 138 - Магический метод `__iter__`

3. Добавляем магический метод `__len__` и создаем метод `save`.



```
1 def __len__(self):
2     return sum(item['quantity'] for item in self.cart.values())
3
4 def save(self):
5     self.session.modified = True
```

Рисунок 139 - Магический метод `__len__` и метод `save`

4. Создаем метод `add` и `remove`, чтобы добавлять и удалять товары.

```
1  def add(self, book, quantity=1, update_quantity=False):
2      book_id = str(book.id)
3
4      if book_id not in self.cart:
5          self.cart[book_id] = {'quantity' : 0, 'price': book.price}
6
7      if update_quantity:
8          self.cart[book_id]['quantity'] = quantity
9
10     else:
11         self.cart[book_id]['quantity'] += quantity
12     self.save()
13
14 def remove(self, book):
15     book_id = str(book.id)
16
17     if book_id in self.cart:
18         if self.cart[book_id]['quantity'] > 1:
19             self.cart[book_id]['quantity'] -= 1
20         else:
21             del self.cart[book_id]
22     self.save()
```

Рисунок 140 - Методы add и remove

5. Создаем методы get_total_price и clear.

```
1  def get_total_price(self):
2
3      return sum(int(item['price']) * int(item['quantity'])) for item in self.cart.values())
4
5
6
7  def clear(self):
8      del self.session[self.CART_SESSION_ID]
9      self.save()
```

Рисунок 141 - Методы get_total_price и clear

6. Создаем функций с помощью, которых будем принимать запросы с id товара.



```
1  from django.shortcuts import render
2  from .cart import CartSession
3  from django.http import HttpResponseRedirect
4  from book.models import Book
5  from django.shortcuts import render, redirect, get_object_or_404
6  from django.urls import reverse
7
8  def cart_add(request: HttpResponseRedirect, book_id):
9      cart = CartSession(request.session)
10     book = get_object_or_404(Book, id=book_id)
11     cart.add(book=book)
12
13     return redirect(reverse('cart_detail'))
14
15 def cart_remove(request: HttpResponseRedirect, book_id):
16     cart = CartSession(request.session)
17     book = get_object_or_404(Book, id=book_id)
18     cart.remove(book=book)
19     return redirect(reverse('cart_detail'))
20
21 def cart_detail(request: HttpResponseRedirect):
22     cart = CartSession(request.session)
23     return render(request, 'cart_detail.html', context={
24         'cart': cart
25     })
```

Рисунок 142 - Функции для добавление, удаления и чтение товаров в корзине

Данных функция просто используем класс CartSession и передаем аргумент request.session.

7. В карточке товара создаем кнопку, чтобы добавить товар в корзину



```
1 <a href="{% url 'cart_add' book.id %}"> В корзину</a>
```

Рисунок 143 - Ссылка на добавление товара в корзину

8. Создаем страницу корзины cart_detail.html

```
1  {% extends "base.html" %}  
2  
3  
4  {% block content %}  
5      <div class="cart__body">  
6          {% for item in cart %}  
7  
8              <div class="item__cart">  
9                  Количество: {{item.quantity}}  
10                 Цена: {{ item.price }}  
11  
12                   
13                 <a href='{% url "cart_remove" item.book.id %}'>Удалить</a>  
14  
15  
16  
17          </div>  
18  
19      {% endfor %}  
20      </div>  
21      Сумма корзины:{{ cart.get_total_price }}  
22  {% endblock %}
```

Рисунок 144 - Файл detail_cart.html

Не забывайте что в шаблон передали объект CartSession

9. Теперь создадим контекст процессор, который будет отображать на всех страницах количество товаров корзине. В основной приложении(app) создаем файл с context_processors.py



```
1 from cart.cart import CartSession
2
3
4 def cart(request):
5
6     return {'cart' : CartSession(request.session), }
```

Рисунок 145 - Файл context_processors.py

10. Заходим в settings.py, и в переменной TEMPLATES



```
1 TEMPLATES = [
2     {
3         'BACKEND': 'django.template.backends.django.DjangoTemplates',
4         'DIRS': [TEMPLATES_DIR],
5         'APP_DIRS': True,
6         'OPTIONS': {
7             'context_processors': [
8                 'django.template.context_processors.debug',
9                 'django.template.context_processors.request',
10                'django.contrib.auth.context_processors.auth',
11                'django.contrib.messages.context_processors.messages',
12
13
14                'app.context_processors.cart',
15
16            ],
17        },
18    },
19]
```

Рисунок 146 - Переменная TEMPLATES

11. Ссылка корзина и отображения количества товаров в корзине



```
1  <li><a href="{% url 'cart_detail' %}">Корзина {{cart |length}}</a></li>
```

Рисунок 147 - Отображения количества товаров

ЗАДАНИЕ

1. Сделай чтобы при добавление товара количество товаров увеличивалось без перезагрузки страницы, а также пропадала кнопка «в корзину»
2. Добавьте уменьшения и увеличение товаров в корзине
3. Сделай кнопку, которая очищает корзину полностью

Контрольные вопросы

1. Для чего нужны контекст процессор?
2. Где можно хранить сессию?

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №15

TabularInline админ и создание заказов

ПРАКТИЧЕСКАЯ ЧАСТЬ

1. Создаем приложение order, в котором будет весь функционал связанный с заказами.

Создаем модели Order и OrderItem:

```
● ● ●

1  class PaymentStatus(models.TextChoices):
2      PENDING = "На рассмотрении"
3      PROCESSED = "В обработке"
4      SHIPPED = "Отправлен"
5      DELIVERED = "Доставлены"
6
7  class Order(models.Model):
8      customer_user = models.ForeignKey(User, on_delete=models.CASCADE)
9      customer_email = models.EmailField()
10     order_date = models.DateTimeField(auto_now_add=True)
11     status = models.CharField(max_length=50, choices=PaymentStatus)
12     paid = models.BooleanField(default=False)
13
14
15     def __str__(self):
16         return f"Order {self.id} for {self.customer_user.username}"
```

Рисунок 148 - Модель Order

```
● ● ●

1  class OrderItem(models.Model):
2      order = models.ForeignKey(Order, related_name='items', on_delete=models.CASCADE)
3      book = models.ForeignKey(Book, on_delete=models.CASCADE)
4      quantity = models.IntegerField()
5
6      def __str__(self):
7          return f"{self.quantity} of {self.book} for {self.order}"
```

Рисунок 149 - Модель OrderItem

2. Создаем форму для оформление заказа

```
1 class OrderForm(forms.ModelForm):  
2  
3     class Meta:  
4         model = Order  
5         fields = ['customer_email']
```

Рисунок 150 - Форма для оформление заказа

3. В файле views.py пишем логику оформление заказа. Оформление заказа будет происходить на страницы корзины, а данные будут брать с сессии корзины.

```
1 @login_required(login_url='login')  
2 def create_order(request: HttpRequest):  
3     cart = CartSession(request.session)  
4     if request.method == "POST":  
5         form = OrderForm(request.POST)  
6         if form.is_valid():  
7             order = form.save(commit=False)  
8             order.customer_user = request.user  
9             order.save()  
10            for item_cart in cart:  
11                OrderItem.objects.create(order=order, book=item_cart['book'], quantity=item_cart['quantity']).save()  
12            cart.clear()  
13            return redirect(reverse('profile'))  
14        else:  
15            form = OrderForm()
```

Рисунок 151 - Функция для отображение заказа

4. Вставляем форму оформление заказа на страницу корзины

```
1 <div class="">
2     <div class="">
3         <h2>Оформление заказа</h2>
4     </div>
5     <form action="{% url 'create_order' %}" method="post">
6
7         {% csrf_token %}
8         {{ orderForm.as_p }}
9         <button type="submit">Заказа</button>
10    </form>
11 </div>
```

Рисунок 152 - Форма оформление заказа

5. Заходим в файл admin.py, и используем класс TabularInline, чтобы отобразить связи в админ-панели на одной страницы.

```
1 class OrderItemInline(admin.TabularInline):
2     model = OrderItem
3     extra = 0
4
5 @admin.register(Order)
6 class OrderAdmin(admin.ModelAdmin):
7     list_display = ('customer_user', 'customer_email', 'order_date', 'status')
8     search_fields = ('customer_user', 'customer_email', 'status')
9     list_filter = ('status', 'order_date')
10    inlines = [OrderItemInline]
```

Рисунок 153 - Файл admin.py для заказов

6. Теперь в админ-панели будет показываться заказы с товарами на одной странице

Изменить order

ИСТОРИЯ

Order 2 for asdasd

Customer user: asdasd   

Customer email: asdasdy@gmail.com

Status: Processed 

Paid

ORDER ITEMS

BOOK

1 of Book object (4) for Order 2 for asdasd

Book object (4)   

QUANTITY

УДАЛИТЬ?

1 



1 of Book object (3) for Order 2 for asdasd

Book object (3)   

1 



 Добавить еще один Order item

СОХРАНИТЬ

Сохранить и добавить другой объект

Сохранить и продолжить редактирование

Удалить

Рисунок 154 - Админ-панель с заказов

ЗАДАНИЕ

1. Отобразить заказы пользователя с товарами на странице профиля (для этого используйте класс Q и select_related)

ПРАКТИЧЕСКОЕ ЗАДАНИЕ №16

Тестирование ЦЕЛЬ ЗАНЯТИЯ

1. Создание тестов

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Тестирование является неотъемлемой частью процесса разработки программного обеспечения. Оно помогает выявить ошибки и дефекты в коде до того, как они достигнут конечного пользователя. В Django, одном из самых популярных фреймворков для веб-разработки на Python, предусмотрены мощные инструменты для написания и выполнения тестов.

Виды тестирования

В Django можно проводить различные виды тестирования:

1. **Модульные тесты (Unit Tests)**: Тестируют отдельные компоненты или функции кода.
2. **Функциональные тесты (Functional Tests)**: Проверяют работу приложения с точки зрения пользователя.
3. **Интеграционные тесты (Integration Tests)**: Проверяют взаимодействие между различными компонентами системы.
4. **Системные тесты (System Tests)**: Проверяют всю систему целиком, включая внешние зависимости.

Средства для тестирования в Django

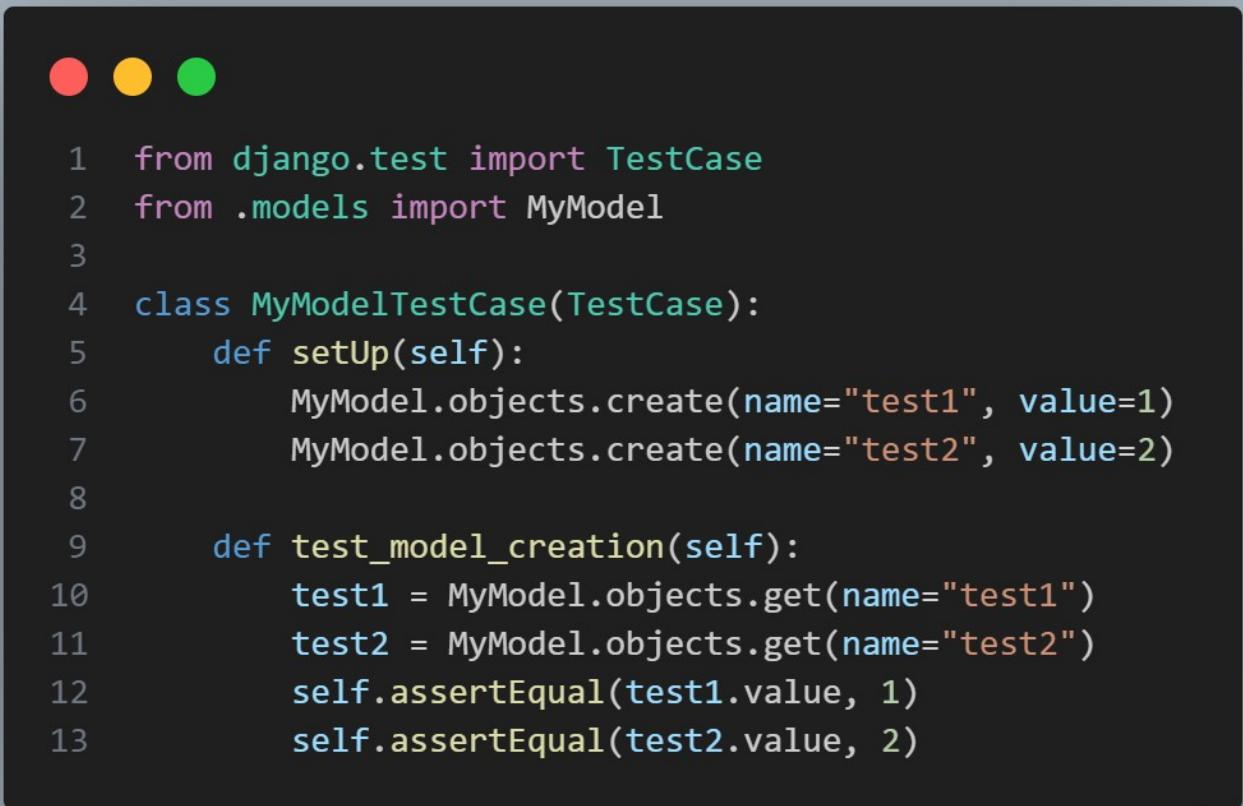
Django использует встроенные средства и библиотеки для написания тестов, в том числе:

- **unittest**: Библиотека Python для создания и выполнения тестов. Django использует расширение этой библиотеки.
- **TestCase**: Класс, предоставляемый Django, который расширяет возможности unittest.TestCase.
- **Client**: Класс для симуляции HTTP-запросов и тестирования представлений (views).

Основы написания тестов в Django

Для начала тестирования в Django необходимо создать тестовые файлы и классы. Обычно они располагаются в файле `tests.py` внутри каждого приложения.

Пример теста модели



```
1  from django.test import TestCase
2  from .models import MyModel
3
4  class MyModelTestCase(TestCase):
5      def setUp(self):
6          MyModel.objects.create(name="test1", value=1)
7          MyModel.objects.create(name="test2", value=2)
8
9      def test_model_creation(self):
10         test1 = MyModel.objects.get(name="test1")
11         test2 = MyModel.objects.get(name="test2")
12         self.assertEqual(test1.value, 1)
13         self.assertEqual(test2.value, 2)
```

Рисунок 155 - Тест модели

В этом примере:

- Метод `setUp` создаёт тестовые данные перед каждым тестом.
- Метод `test_model_creation` проверяет корректность создания и хранения данных.

Пример теста представления (view)

```
1  from django.test import TestCase, Client
2
3  class MyViewTestCase(TestCase):
4      def setUp(self):
5          self.client = Client()
6
7      def test_index_view(self):
8          response = self.client.get('/')
9          self.assertEqual(response.status_code, 200)
10         self.assertContains(response, "Welcome to the index page")
11
```

Рисунок 156 - Тесты views

В этом примере:

- Класс `Client` используется для имитации HTTP-запросов.
- Метод `test_index_view` проверяет корректность ответа на GET-запрос к главной странице.

Запуск тестов

Для запуска тестов в Django используется команда:

```
python manage.py test
```

ПРАКТИЧЕСКАЯ ЧАСТЬ

В этой практической напишем тест для приложения account views.

1. Заходим в файл views.py. Создаем класс TestCaseViewAccount и наследуемся от TestCase. Создаем метод test_register_view (методы должны начинаться test)

The screenshot shows a code editor with a dark theme. At the top left, there are three colored circular icons: red, yellow, and green. The code itself is a Python unittest test case:

```
1 from django.test import TestCase
2 from .forms import RegisterForm
3 from django.urls import reverse
4 from django.contrib.auth.models import User
5
6 class TestCaseViewAccount(TestCase):
7
8
9     def setUp(self) -> None:
10         self.form_data ={
11             'username' : 'ASDFGHJK',
12             'email' : 'asdasdasd@gmail.com',
13             'password1' : 'Qwewqesdfafsd1234435',
14             'password2' : 'Qwewqesdfafsd1234435',
15         }
16     def test_register_view(self):
17
18         RegisterForm(data=self.form_data)
19         self.client.post(reverse('register'), data=self.form_data)
20         user = User.objects.filter(username=self.form_data['username']).exists()
21         self.assertTrue(user)
```

Рисунок 157 - Тест на регистрацию

В test_register_view проверяет зарегистрировался ли пользователь. При тестировании создается отдельная БД .

2. После написание теста, пишем команду py manage.py test

The screenshot shows a terminal window with the following output:

```
Found 1 test(s).
Creating test database for alias 'default',...
System check identified no issues (0 silenced).

.
-----
Ran 1 test in 0.287s

OK
Destroying test database for alias 'default'...
```

Рисунок 158 - Успешное проверка тестов

Контрольные вопросы

1. Для чего нужны тестировать веб-приложение?