Базовые представления на основе классов

Class based views

Базовые представления на основе классов

Базовые представления на основе классов абстрагируют общие шаблоны до такой степени, что вам даже не нужно писать код на Python, чтобы создать приложение. Например, общие представления ListView и DetailView абстрагируют понятия «отобразить список объектов» и «отобразить страницу с деталями для определенного типа объектов» соответственно.

Django предлагает готовые «шаблоны» для типичных задач:

- TemplateView — для простого отображения HTML-шаблона.
- ListView — для вывода списка объектов из базы данных.
- DetailView — для страницы с детальной информацией об одном объекте.
- CreateView / UpdateView / DeleteView — для работы с формами создания, изменения и удаления данных.

TemplateView

views.py

```
from django.views.generic.base import TemplateView
class HomePageView(TemplateView):
    template_name = "books/home.html"
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        return context
```

urls.py

```
from django.urls import path
from books.views import *
app_name = "books"
urlpatterns = [
    path("", HomePageView.as_view(), name="home"),
]
```

TemplateView

Самый прямой способ использования универсальных представлений — создавать их непосредственно в URLconf. Если вы изменяете лишь несколько атрибутов в представлении на основе класса, вы можете передать их **as_view()** непосредственно в вызов метода

urls.py

```
from django.urls import path

from django.views.generic import TemplateView

app_name = "books"

urlpatterns = [
    path("about/", TemplateView.as_view(template_name="books/about.html"),
 name="about"),
,
]
```

ListView

Класс **ListView** в Django — это встроенный инструмент для автоматического отображения списка объектов из базы данных. Он избавляет от необходимости вручную писать запросы к БД и передавать их в контекст.

Базовый пример реализации на примере списка книг (Модель Book):

views.py

```
from django.views.generic import ListView
from .models import Book
class BookListView(ListView):
    model = Book
    # По умолчанию ищет шаблон: <app_name>/book_list.html
    # Данные в шаблоне доступны под именем 'object_list' или 'book_list’
```

urls.py

```
from django.urls import path
from .views import BookListView
urlpatterns = [
    path('books/', BookListView.as_view(), name='book-list'),
]
```

ListView

Шаблон (book_list.html)

```
<h1>Список книг</h1>
<ul>
  {% for book in book_list %}
    <li>{{ book.title }} — {{ book.author }}</li>
  {% empty %}
    <li>Книг пока нет.</li>
  {% endfor %}
</ul>
```

ListView

Вы можете изменить стандартное поведение, переопределив атрибуты класса:

- template_name: Позволяет задать произвольный путь к шаблону (например, 'myapp/custom_list.html').
- context_object_name: Позволяет переименовать переменную в шаблоне (например, context_object_name = 'my_books').
- paginate_by: Включает пагинацию. Например, paginate_by = 10 будет показывать по 10 объектов на странице.
- get_queryset(): Метод для динамической фильтрации или сортировки списка.
- get_context_data(): Позволяет передать в шаблон что-то кроме списка объектов

ListView

```python
class AdvancedBookListView(ListView):
    model = Book
    template_name = 'books/book_list.html'
    context_object_name = 'books'
    paginate_by = 10  # Пагинация: 10 книг на страницу

    def get_queryset(self):
        queryset = super().get_queryset()
        # Фильтрация по категории из URL (например, /books/scifi/)
        category = self.kwargs.get('category')
        if category:
            queryset = queryset.filter(category__slug=category)
        # Поиск по строке из GET-параметра (например, ?q=django)
        query = self.request.GET.get('q')
        if query:
            queryset = queryset.filter(title__icontains=query)
        return queryset.order_by('-created_at')

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['total_count'] = Book.objects.count()
        context['search_query'] = self.request.GET.get('q', '')
        return context

path('books/', AdvancedBookListView.as_view(), name='book-list'),
path('books/<slug:category>/', AdvancedBookListView.as_view(), name='book-list-by-category'),
```

ListView

```
<form method="get">
    <input type="text" name="q" value="{{ search_query }}" placeholder="Поиск книг...">
    <button type="submit">Найти</button>
</form>

<p>Всего книг в базе: {{ total_count }}</p>
<ul>
    {% for book in books %}
        <li>{{ book.title }}</li>
    {% empty %}
        <li>Ничего не найдено.</li>
    {% endfor %}
</ul>

{% if is_paginated %}
    <nav>
        {% if page_obj.has_previous %}
            <a href="?page={{ page_obj.previous_page_number }}{% if search_query %}&q={{ search_query }}{% endif
  %}">Назад</a>
        {% endif %}
        <span>Страница {{ page_obj.number }} из {{ paginator.num_pages }}</span>
        {% if page_obj.has_next %}
            <a href="?page={{ page_obj.next_page_number }}{% if search_query %}&q={{ search_query }}{% endif
  %}">Вперед</a>
        {% endif %}
    </nav>
{% endif %}
```

DetailView

DetailView в Django используется для отображения одного конкретного объекта из базы данных. Он автоматически извлекает объект по первичному ключу (pk) или slug, переданному через URL.

views.py

```
from django.views.generic import DetailView
from .models import Book
class BookDetailView(DetailView):
    model = Book
```

urls.py

```
from django.urls import path
from .views import BookDetailView
urlpatterns = [
    path('book/<int:pk>/', BookDetailView.as_view(), name='book-detail'),
]
```

book_detail.html

```
<h1>{{ book.title }}</h1>
<p>{{ book.description }}</p>
```

DetailView

- model: Модель, из которой берется объект.
- template_name: Путь к вашему .html файлу.
- get_context_data(self, **kwargs): Используется для передачи дополнительных данных в шаблон (формы, списки других объектов, текущее время и т.д.).
- context_object_name: Имя переменной в шаблоне. По умолчанию это либо object, либо имя модели в нижнем регистре (например, book).
- queryset: Если нужно ограничить выборку (например, Book.objects.filter(is_active=True)). Если указан queryset, атрибут model можно не писать.
- get_object(self, queryset=None): Для реализации сложной логики поиска объекта.
- get_queryset(self): Позволяет динамически менять условия фильтрации. Например, администратор видит все книги, а обычный пользователь — только опубликованные.

DetailView

```python
class BookDetailView(DetailView):
    model = Book
    template_name = 'books/book_detail.html'
    context_object_name = 'book'  # Имя переменной в шаблоне (по умолчанию 'object')

    def get_queryset(self):
        """Фильтрация: показывать только опубликованные книги"""
        return Book.objects.filter(is_published=True)

    def get_context_data(self, **kwargs):
        """Дополнительные данные: например, список похожих книг"""
        context = super().get_context_data(**kwargs)
        # Добавляем в контекст 3 случайных книги того же автора
        context['similar_books'] = Book.objects.filter(
            author=self.object.author).exclude(id=self.object.id)[:3]
        return context

    def get(self, request, *args, **kwargs):
        """Логика при просмотре: например, счетчик просмотров"""
        response = super().get(request, *args, **kwargs)
        # Увеличиваем счетчик просмотров у объекта
        self.object.views_count += 1
        self.object.save()
        return response
```

Шаблоны

Наследование шаблонов

Наследование шаблонов позволяет создать базовый «скелетный» шаблон, который содержит все общие элементы сайта и определяет блоки, которые дочерние шаблоны могут переопределить.
Реализация наследования шаблонов основана на двух базовых тегах:

- `block` – создает блок, в который могут быть вставлены различные элементы HTML-страницы;
- `extend` – расширяет текущий шаблон.

Любой участок шаблона можно обернуть в тег `block`, при этом блоку дается имя, например:

```
{% block content %}
        тело блока
{% endblock %}
```

Любой родительский шаблон можно расширить путем создания на него ссылки из дочернего шаблона, например:

```
{% extend "base.html" %}
```

Базовый шаблон base.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <header>
        <nav>
            <a href="/">Главная</a> | <a href="/books/">Книги</a>
        </nav>
    </header>
    <main>
        {% block content %}{% endblock %}
    </main>
    <footer>
        <p>&copy; 2026 </p>
    </footer>
</body>
</html>
```

Дочерний шаблон

```
{% extends "base.html" %} <!-- Наследуемся от базового шаблона -->


{% block title %}Список всех книг{% endblock %} <!-- Заполняем блок заголовка -->


{% block content %} <!-- Заполняем основной контент -->
    <h1>Наши книги</h1>
    <ul>
        {% for book in object_list %}
            <li>{{ book.title }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

Включение шаблонов

Имеется возможность произвести включение произвольного шаблона в состав другого шаблона. Для этого используется тег шаблонизатора `include`:

```
{% include <путь к включаемому шаблону> [with <дополнительные переменные
контекста включаемого шаблона>] [only] %}
```

Включаемый шаблон автоматически получит тот же контекст, что и включающий шаблон. Можно указать дополнительные переменные, которые будут добавлены в контекст включаемого шаблона, записав их в формате `<имя переменной>=<значение переменной>` через пробел. Ключевое слово only предписывает не передавать контекст включающего шаблона включаемому шаблону.

Тег include записывается в том месте кода включающего шаблона, в котором должен быть выведен включаемый шаблон.

```
{% include 'my_apps/name_snippet.html' with greeting='Привет' %}
```

name_snippet.html

```
{{ greeting }}!
```

Обработка статических файлов

В терминологии `Django` статическими называются файлы, отправляемые клиенту как есть: таблицы стилей, графические изображения, аудио- и видеоролики, файлы статических веб-страниц, архивы и т. п.

Обработку статических файлов выполняет подсистема, реализованная во встроенном приложении `django.contrib.staticfiles`. Оно включается в список зарегистрированных приложений уже при создании нового проекта, и, если сайт содержит статические файлы, удалять его оттуда нельзя.

Для работы со статическими файлами (CSS, JS, изображения) в Django используется специальный тег `{% load static %}`. Это гарантирует, что пути к файлам будут строиться правильно, даже если вы измените настройки сервера. Тег `{% load static %}` всегда пишется в самой первой строчке файла.

settings.py

```
STATIC_URL = 'static/'
STATICFILES_DIRS = [BASE_DIR / "static"]
```

Где ищутся статические файлы?

Как это работает при разработке (DEBUG = True)

Когда вы запрашиваете файл через {% static 'css/style.css' %}, Django проверяет два основных места:

- В папках приложений (AppDirectoriesFinder):
  Django обходит все установленные приложения в INSTALLED_APPS и ищет в них подпапку static.
  Пример: my_app/static/css/style.css.
- В общих папках (FileSystemFinder):
  Django смотрит в список путей, которые вы указали вручную в настройке STATICFILES_DIRS в settings.py. Обычно там указывают общую папку проекта.
  Пример: STATICFILES_DIRS = [BASE_DIR / "static"].

Приоритет поиска

Django берет первый найденный файл. Если файл с одинаковым именем есть и в корневой папке, и внутри приложения, Django возьмет тот, чей «искатель» стоит выше в настройках (по умолчанию сначала общие папки из STATICFILES_DIRS, затем папки приложений).

Где ищутся статические файлы?

Как это работает в продакшне (DEBUG = False)
На реальном сервере Django не должен сам раздавать статические файлы (это медленно). Происходит следующее:

- Вы запускаете команду python manage.py collectstatic.
- Django обходит все папки (приложения и STATICFILES_DIRS) и копирует все найденные файлы в одну общую директорию, указанную в STATIC_ROOT.
- Веб-сервер (например, Nginx) настраивается так, чтобы отдавать файлы напрямую из этой папки STATIC_ROOT.

Золотое правило именования
Чтобы файлы разных приложений не конфликтовали (например, у двух приложений есть style.css), внутри папки static принято создавать подпапку с именем приложения:
my_app/static/my_app/css/style.css

my_apps/static/my_apps/style.css

```
body{
    background-color: papayawhip;
}
```

my_apps/templates/my_apps/index.html

```
{% load static %}
<link rel="stylesheet" href="{% static 'my_apps/style.css' %}">
```