

Лекция 4.

Работа с формами



На этой лекции мы

1. Узнаем о формах Django
2. Разберемся в создании классов форм
3. Изучим поля и виджеты форм
4. Узнаем о обработке данных из формы
5. Разберемся в сохранении файлов

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Узнали о представлениях Django
2. Разобрались в работе диспетчера URL
3. Изучили шаблоны и передачу контекста в них
4. Узнали о условиях, циклах и наследовании шаблонов
5. Объединили модели, представления, шаблоны и маршруты

План лекции

[На этой лекции мы](#)

[Краткая выжимка, о чём говорилось в предыдущей лекции](#)

[План лекции](#)

[Подробный текст лекции](#)

[Что такое формы в Django?](#)

[Вместо старта](#)

[Создание форм](#)

[Создаём класс формы](#)

[Представление для формы](#)

[Прописываем url](#)

[Отрисовка шаблона](#)

[Доработка шаблона](#)

[Поля и виджеты форм](#)

[Поля форм](#)

[Представление](#)

[Маршрут](#)

[Шаблон](#)

[Виджеты форм](#)

[Ручное изменение типа поля](#)

[Обработка данных форм](#)

[Пользовательская валидация данных с помощью метода clean\(\)](#)

[Сохранение формы в базу данных](#)

[Модель данных](#)

[Представление](#)

[Шаблон](#)

[Маршрут](#)

[Сохранение изображений \(файлов\)](#)

[Форма forms.py](#)

[Настройка settings.py](#)

[Представление views.py](#)

[Маршрут urls.py](#)

[Шаблон templates/myapp4](#)

[Итоги](#)

[Вывод](#)

[Домашнее задание](#)

Подробный текст лекции

Что такое формы в Django?

Формы в Django — это инструмент, который позволяет создавать и обрабатывать HTML-формы. Формы используются для сбора данных от пользователей и отправки

их на сервер для дальнейшей обработки.

Определение формы в Django начинается с создания класса формы, который наследуется от класса `forms.Form`. В этом классе определяются поля формы, их типы, валидация и другие атрибуты. Поля формы могут быть текстовыми, числовыми, выбором из списка и т.д.

Плюсы использования форм в Django заключаются в том, что они позволяют упростить процесс сбора данных от пользователей и обработки их на сервере. Формы автоматически проверяют данные на корректность, что позволяет избежать ошибок при обработке данных. Кроме того, формы могут быть переиспользованы в разных частях приложения, что упрощает разработку и поддержку кода.

В целом, использование форм в Django - это удобный и эффективный способ сбора и обработки данных от пользователей, который позволяет упростить разработку и поддержку веб-приложений. Однако, при использовании форм необходимо учитывать их ограничения и принимать меры для защиты от возможных уязвимостей.

Вместо старта

Если вы создавали новое приложение для каждого занятия, выполните команды:

```
>cd myproject
>python manage.py startapp myapp4
```

Отлично! Новое приложение создано в проекте. Сразу подключим его в настройках `setting.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    ...
    'myapp4',
]
```

Всё готово к началу изучения форм на практике.

Создание форм

В Django для создания форм используются классы, которые наследуются от класса `forms.Form`.

Создаём класс формы

Рассмотрим пример определения формы для ввода данных о пользователе. Для этого создадим файл `forms.py` в приложении:

```
from django import forms

class UserForm(forms.Form):
    name = forms.CharField(max_length=50)
    email = forms.EmailField()
    age = forms.IntegerField(min_value=0, max_value=120)
```

В данном примере создается класс `UserForm`, который наследуется от класса `forms.Form`. В классе определены три поля формы: `name`, `email` и `age`. Каждое поле представлено отдельным классом, который определяет тип поля и его атрибуты.

- Поле `name` определено с помощью класса `CharField`, который представляет текстовое поле. Аргумент `max_length` указывает максимальную длину текста, которую можно ввести в поле.
- Поле `email` определено с помощью класса `EmailField`, который представляет поле для ввода email-адреса. Этот класс автоматически проверяет введенный адрес на корректность.
- Поле `age` определено с помощью класса `IntegerField`, который представляет числовое поле. Аргументы `min_value` и `max_value` указывают минимальное и максимальное значение, которое можно ввести в поле.

Представление для формы

Следующий этап — использовать представление для перевода формы из класса в видимый пользователем HTML, а также для обработки данных, которые пользователь введёт в форму и отправит на сервер.

Для вывода формы по GET запросу и обработки данных по POST запросу в Django можно использовать следующий код:

```

import logging

from django.shortcuts import render
from .forms import UserForm

logger = logging.getLogger(__name__)

def user_form(request):
    if request.method == 'POST':
        form = UserForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            age = form.cleaned_data['age']
            # Делаем что-то с данными
            logger.info(f'Получили {name=}, {email=}, {age=}.'.)
        else:
            form = UserForm()
    return render(request, 'myapp4/user_form.html', {'form':
form})

```

В данном случае мы импортируем модуль `render` для рендеринга шаблона, а также импортируем нашу форму `UserForm`. Далее определяем функцию `user_form`, которая будет обрабатывать запросы на адрес `/user_form/`.

Если запрос пришел методом `POST`, то мы создаем экземпляр формы `UserForm` с переданными данными из запроса. Если форма проходит валидацию (все поля заполнены корректно), то мы получаем данные из формы и можем с ними работать. Если запрос пришел методом `GET`, то мы просто создаем пустой экземпляр формы `UserForm` и передаем его в шаблон `user_form.html`.

Прописываем url

В шаблоне `user_form.html` мы можем вывести нашу форму с помощью тега `{{ form }}`. Также можно добавить кнопку `Submit` для отправки формы. Но ничего этого не получится, если пользователь не имеет доступа к форме. Переходим в `urls.py` проекта и подключаем новое приложение:

```

from django.contrib import admin
from django.urls import path, include

```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    ...
    path('les4/', include('myapp4.urls')),
]
```

А теперь прописываем маршрут для связи представления с url-адресом. Для этого создаём urls.py внутри каталога приложения:

```
from django.urls import path
from .views import user_form

urlpatterns = [
    path('user/add/', user_form, name='user_form'),
]
```

Форма будет доступна по адресу <http://127.0.0.1:8000/les4/user/add/> Остался финальный шаг.

Отрисовка шаблона

В представлении мы используем функцию `render()` для пробрасывания формы в шаблон `user_form.html`. Создадим каталог `templates` внутри приложения, а в нём каталог с именем приложения. В нашем случае каталог `myapp4/`.



Внимание! Как и в прошлых занятиях мы не используем вёрстку, чтобы сосредоточить внимание на Django и бекенд разработке.

Файл `base.html` в каталоге `templates` проекта:

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}Формы{% endblock %}</title>
</head>
<body>
    {% block content %}
```

```
        Контент скоро появится...
    {% endblock %}
</body>
</html>
```

Файл user_form.html в первой итерации. Каталог templates/myapp4 приложения:

```
{% extends 'base.html' %}

{% block content %}
    {{ form }}
{% endblock %}
```

На первый взгляд кажется, что мы сделали всё необходимое. Переходим по адресу <http://127.0.0.1:8000/les4/user/add/> и смотрим на содержимое страницы. Обычно клавиша F12 для доступа к коду. В нашем случае получилась следующая страница:

```
<!DOCTYPE html>
<html lang="ru"><head>
    <meta charset="UTF-8">
    <title>Формы</title>
</head>
<body wfd-invisible="true">
    <label for="id_name">Name:</label>
    <input type="text" name="name" maxlength="50" required=""
id="id_name">
    <label for="id_email">Email:</label>
    <input type="email" name="email" required="" id="id_email">
    <label for="id_age">Age:</label>
    <input type="number" name="age" min="0" max="120" required=""
id="id_age">
</body>
</html>
```

Класс UserForm был преобразован в набор полей label и input. Но тег form и кнопка отправки отсутствуют. За их добавление, а также за шифрование данных отвечает разработчик.


Доработка шаблона

Чтобы мы действительно увидели форму ввода, внесём правки в шаблон `user_form.html`

```
{% extends 'base.html' %}

{% block content %}
<form action="" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Отправить">
</form>
{% endblock %}
```

Мы добавили тег формы указав что при нажатии кнопки отправить нужно использовать тот же url адрес для отправки данных и метод POST для пересылки. Так же был добавлен тег `csrf_token`. Он обеспечивает защиту данных формы от изменений злоумышленниками. Так называемая защита от CSRF атак. Кроме того вручную добавлена кнопка отправки и закрывающий тег формы.

 **Важно!** Четыре добавленные строки являются обязательными для правильного отображения любой формы через шаблоны Django.

Подведём предварительный итог. Для работы с формами необходимо создать класс формы в файле `forms.py`. Далее создаётся представление, которое отправляет экземпляр формы в шаблон. Сам шаблон должен отрисовать html теги формы и передать `csrf_token` и экземпляр формы. При этом стоит помнить про настройку маршрутов.

Поля и виджеты форм

Разберемся подробнее в том какие поля мы можем создавать внутри класса формы. По сути каждое поле — это экземпляр класс `<Name>Field` из модуля `forms`, где `<Name>` - имя класса поля.

Поля форм

Перечислим некоторые из наиболее популярных классов Field в Django:

- CharField — используется для создания текстовых полей, таких как имя, фамилия, электронная почта и т.д.
- EmailField — используется для создания поля электронной почты.
- IntegerField — используется для создания поля для ввода целых чисел.
- FloatField — используется для создания поля для ввода чисел с плавающей точкой.
- BooleanField — используется для создания поля флажка.
- DateField — используется для создания поля даты.
- DateTimeField — используется для создания поля даты и времени.
- FileField — используется для создания поля для загрузки файла.
- ImageField — используется для создания поля для загрузки изображения.
- ChoiceField — используется для создания выпадающего списка с выбором одного из нескольких вариантов.

Быстрее всего разобраться в различиях полей получится на примере. Создадим форму для демонстрации разнообразия полей в файле forms.py:

```
import datetime

from django import forms

...
class ManyFieldsForm(forms.Form):
    name = forms.CharField(max_length=50)
    email = forms.EmailField()
    age = forms.IntegerField(min_value=18)
    height = forms.FloatField()
    is_active = forms.BooleanField(required=False)
    birthdate = forms.DateField(initial=datetime.date.today)
    gender = forms.ChoiceField(choices=[('M', 'Male'), ('F', 'Female')])
```

Мы создали форму на семь различных полей. При этом указали несколько параметров:

- текстовое поле ограничено 50 символами.
- возраст должен быть не меньше 18
- для поля is_active прописали отсутствие “галочки” по умолчанию. Обязательно прописывать для логического поля параметр required

- при вводе дня рождения нам заранее демонстрируется текущая дата
- выбор пола показывается как поле с выбором из двух вариантов. При этом пользователь видит Male и Female, а в переменную сохраняются М или F.

Для работы с формой надо внести дополнения в код.

Представление

В views.py допишем новое представление

```
import logging

from django.shortcuts import render
from .forms import UserForm, ManyFieldsForm

logger = logging.getLogger(__name__)

...
def many_fields_form(request):
    if request.method == 'POST':
        form = ManyFieldsForm(request.POST)
        if form.is_valid():
            # Делаем что-то с данными
            logger.info(f'Получили {form.cleaned_data=}.')
        else:
            form = ManyFieldsForm()
            return render(request, 'myapp4/many_fields_form.html',
                          {'form': form})
```

Ничего нового мы не добавили. Стандартный вывод пустой формы при GET запросе и формирование формы с данными при POST запросе с последующей проверкой и возможной обработкой.

Маршрут

В urls.py допишем новый маршрут

```
from django.urls import path
from .views import user_form, many_fields_form

urlpatterns = [
    path('user/add/', user_form, name='user_form'),
    path('forms/', many_fields_form, name='many_fields_form'),
]
```

Теперь по адресу <http://127.0.0.1:8000/les4/forms/> мы можем увидеть нашу форму.

Шаблон

Финальный штрих — добавление шаблона `many_fields_form.html`

```
{% extends 'base.html' %}

{% block title %}Демонстрация полей формы{% endblock %}

{% block content %}
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Отправить">
</form>
{% endblock %}
```

В отличие от прошлого шаблона был добавлен вызов метода `as_p` через точку после переменной `form`. Так мы отрисовываем html файл с выводом каждого поля формы как отдельный абзац.

Виджеты форм

В Django для работы с формами используются виджеты, которые определяют внешний вид и поведение полей формы на странице. В Django предусмотрены встроенные виджеты, такие как `TextInput`, `Select`, `CheckboxInput` и др., которые можно использовать для создания различных типов полей формы.

Вот некоторые из наиболее популярных классов виджетов в Django:

- `TextInput` — используется для создания текстового поля ввода.
- `EmailInput` — используется для создания поля ввода электронной почты.
- `PasswordInput` — используется для создания поля ввода пароля.
- `NumberInput` — используется для создания поля ввода чисел.
- `CheckboxInput` — используется для создания флажка.
- `DateInput` — используется для создания поля ввода даты.
- `DateTimeInput` — используется для создания поля ввода даты и времени.
- `FileInput` — используется для создания поля загрузки файла.
- `Select` — используется для создания выпадающего списка с выбором одного из нескольких вариантов.

- RadioSelect — используется для создания списка радиокнопок.
- Textarea — используется для создания многострочного текстового поля ввода.

Доработаем пример формы из прошлого раздела, Создадим форму аналогичную ManyFieldsForm, но добавим в неё виджеты.

```
import datetime

from django import forms

...
class ManyFieldsFormWidget(forms.Form):
    name = forms.CharField(max_length=50,
                           widget=forms.TextInput(attrs={'class':
'form-control',

'placeholder': 'Введите имя пользователя'}))

    email = forms.EmailField(widget=forms.EmailInput(attrs={'class':
'form-control',

'placeholder': 'user@mail.ru'}))

    age = forms.IntegerField(min_value=18,

widget=forms.NumberInput(attrs={'class': 'form-control'}))

    height = forms.FloatField(widget=forms.NumberInput(attrs={'class':
'form-control'}))

    is_active = forms.BooleanField(required=False,

widget=forms.CheckboxInput(attrs={'class': 'form-check-input'}))

    birthdate = forms.DateField(initial=datetime.date.today,

widget=forms.DateInput(attrs={'class': 'form-control'}))

    gender = forms.ChoiceField(choices=[('M', 'Male'), ('F',
'Female')],

widget=forms.RadioSelect(attrs={'class': 'form-check-input'}))

    message = forms.CharField(widget=forms.Textarea(attrs={'class':
'form-control'}))
```



Внимание! Можно не создавать новые представления, маршруты и шаблон. Достаточно добавить строку импорта класса формы в `views.py` и заменить старую форму на новую в представлении.

В каждом виджете задан атрибут `class` со значениями `form-control`, `form-check-input`, что позволяет использовать стили Bootstrap для оформления полей формы. Атрибут `placeholder`, который определяет текст-подсказку, отображаемый в поле формы до того, как пользователь введет данные.

Для выбора пола использовали виджет `RadioSelect`, теперь вместо раскрывающегося списка у нас перечень значений с возможностью выбрать одно из них.

Кроме того было добавлено поле `message`. Это такое же текстовое поле как и `name`. Но в качестве виджета был выбран `Textarea`, теперь текст сообщения можно вводить в несколько строк.

Ручное изменение типа поля

В большинстве случаев поле формы автоматически используют нужный виджет для отрисовки `html` элемента. Если возникают ситуации, когда отрисовку нужно изменить, используются виджеты. Они позволяют заменить один вид на другой, например обычное поле ввода на текстовую зону (`textarea`). Но бывают ситуации, когда подобного недостаточно. Например в нашем примере даты приходится вводить вручную. Большинство браузеров могут облегчить задачу. Изменим поле с днём рождения на следующую строку:

```
...
birthdate = forms.DateField(initial=datetime.date.today,
widget=forms.DateInput(attrs={'class': 'form-control', 'type':
'date'}))
...
```

Мы вручную поменяли тип поля на “дата”. Теперь браузер рисует кнопку календаря, дату можно выбирать, а не вводить.

Обработка данных форм

Если мы воспользуемся формой из примера выше и попробуем ввести неверные данные, Django автоматически сообщит об этом. Встроенные фильтры не дадут

пройти проверку правильности, метод `form.is_valid()` вернёт ложь. Однако встроенных проверок может быть недостаточно.

Пользовательская валидация данных с помощью метода `clean()`

Мы можем прописать свои методы, которые начинаются со слова `clean_` и далее указать имя поля. Такой метод будет применяться для дополнительной проверки поля на корректность. Рассмотрим пример формы `UserForm` из начала занятия, но добавим пару своих проверок:

```
class UserForm(forms.Form):
    name = forms.CharField(max_length=50)
    email = forms.EmailField()
    age = forms.IntegerField(min_value=0, max_value=120)

    def clean_name(self):
        """Плохой пример. Подмена параметра min_length."""
        name = self.cleaned_data['name']
        if len(name) < 3:
            raise forms.ValidationError('Имя должно содержать не
менее 3 символов')
        return name

    def clean_email(self):
        email: str = self.cleaned_data['email']
        if not (email.endswith('vk.team') or
email.endswith('corp.mail.ru')):
            raise forms.ValidationError('Используйте корпоративную
почту')
        return email
```

В данном примере класс `UserForm`, который наследуется от класса `forms.Form`. В классе определены три поля формы: `name`, `email` и `age`. Для каждого поля определены соответствующие типы данных: `CharField` для поля `name`, `EmailField` для поля `email` и `IntegerField` для поля `age`.

Далее определены два метода `clean_name()` и `clean_age()`, которые осуществляют пользовательскую валидацию данных.

В методе `clean_name()` проверяется длина имени, и если она меньше трех символов, то выбрасывается исключение `ValidationError` с соответствующим сообщением. Это антипаттерн. Мы написали пять строк кода, которые делают тоже самое, что и параметр `min_length=3`.

Для поля `email` встроенные механизмы Django проверяют, что введенный текст похож на электронную почту, с собакой, точкой и т.п. Далее в методе `clean_email()` мы проверяем окончание почты. Если оно не совпадает с одним из корпоративных окончаний, выбрасывается соответствующее сообщение.

Сохранение формы в базу данных

После того как форма заполнена пользователем, отправлена Django, проверена и прошла валидацию данные можно использовать. Обычное использование - сохранение в базу данных.

Модель данных

Начнём с создания модели. Для этого в файле `myapp4/models.py` пропишем следующий код:

```
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    age = models.IntegerField()

    def __str__(self):
        return f'Name: {self.name}, email: {self.email}, age: {self.age}'
```

Обратите внимание, что поля модели очень похожи на поля формы. Но тут используется импорт из `models`, а не `forms`. Впрочем, к похожести мы вернёмся в следующем разделе лекции.

Перед тем как продолжить создадим и применим миграции:

```
python manage.py makemigrations myapp4
python manage.py migrate
```

Форма и шаблон у нас есть, можно переходить к представлениям.

Представление

Доработаем представление из начала урока, чтобы оно сохраняло пользователя в базу данных.

```
import logging

from django.shortcuts import render
from .forms import UserForm
from .models import User

logger = logging.getLogger(__name__)

...
def add_user(request):
    if request.method == 'POST':
        form = UserForm(request.POST)
        message = 'Ошибка в данных'
        if form.is_valid():
            name = form.cleaned_data['name']
            email = form.cleaned_data['email']
            age = form.cleaned_data['age']
            logger.info(f'Получили {name=}, {email=}, {age=}.')
            user = User(name=name, email=email, age=age)
            user.save()
            message = 'Пользователь сохранён'
        else:
            form = UserForm()
            message = 'Заполните форму'
    return render(request, 'myapp4/user_form.html', {'form':
form, 'message': message})
```

Если форма отправлена как POST запрос и проверки данных прошли успешно, получаем переданные данные и создаём экземпляр класса User. Метод user.save() сохраняет запись в таблицу БД.

Шаблон

Дополнительно в код представления добавлена переменная message, которая принимает различные значения в зависимости от этапа обработки данных. Для её отображения необходимо поправить шаблон user_form.html.

```
{% extends 'base.html' %}

{% block content %}
```

```
<h3>{{ message }}</h3>
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_div }}
    <input type="submit" value="Отправить">
</form>
{% endblock %}
```

Маршрут

Финальный этап — подключить представление обработчик, тем самым соединив между собой модель и форму внутри представления.

```
from django.urls import path
from .views import user_form, many_fields_form, add_user

urlpatterns = [
    ...
    path('user/', add_user, name='add_user'),
]
```

Для проверки работы перейдём по адресу <http://127.0.0.1:8000/les4/user/>

Сохранение изображений (файлов)

В финале поговорим про возможности форм Django сохранять изображения пользователя на сервере.



Внимание! Аналогично можно сохранять любые файлы, а не только картинки. Для этого заменяем ImageField на FileField.

Загрузка изображений через форму Django происходит с помощью класса виджета ImageField. Для этого необходимо создать форму, которая будет содержать поле ImageField, а также представление, которое будет обрабатывать данные формы и сохранять загруженное изображение.

Форма forms.py

Пример кода **формы**:

```
class ImageForm(forms.Form):  
    image = forms.ImageField()
```

Настройка settings.py

Теперь позаботимся о том, чтобы Django создал каталог для наших изображений. Перейдём в **settings.py** и пропишем следующие пару констант:

```
...  
MEDIA_URL = '/media/'  
MEDIA_ROOT = BASE_DIR / 'media'  
...
```

Все присланные файлы, в том числе и изображения, будут сохраняться в каталоге media в корне проекта.

Представление views.py

Далее создадим **представление**:

```
...  
from django.core.files.storage import FileSystemStorage  
from django.shortcuts import render  
from .forms import ImageForm  
  
...  
def upload_image(request):  
    if request.method == 'POST':  
        form = ImageForm(request.POST, request.FILES)  
        if form.is_valid():  
            image = form.cleaned_data['image']  
            fs = FileSystemStorage()  
            fs.save(image.name, image)  
        else:  
            form = ImageForm()  
        return render(request, 'myapp4/upload_image.html', {'form':  
form})
```

Если поступил POST запрос, форма заполняется не только из request.POST, но и из request.FILES. Там содержится наше изображение. Если проверка формы успешно завершена, выполняем три действия:

1. Сохраняем изображение в переменной image
2. Создаём экземпляр класса FileSystemStorage для работы с файлами силами Django
3. Просим экземпляр fs сохранить изображение. Метод save принимает имя файла и сам файл

Маршрут urls.py

Пропишем представление в **списке маршрутов**:

```
from django.urls import path
from .views import user_form, many_fields_form, add_user,
upload_image


urlpatterns = [
    ...
    path('upload/', upload_image, name='upload_image'),
]
```

Шаблон templates/myapp4

Финальный этап - создать **шаблон** upload_image.html:

```
{% extends 'base.html' %}

{% block content %}
    <h2>Загрузите изображение</h2>
    <form method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Загрузить</button>
    </form>
{% endblock %}
```

 **Важно!** Чтобы форма отправляет файлы необходимо в теге форм прописать `enctype="multipart/form-data"`. Без этого мы не получим доступ к файлам.

Итоги

Как видите формы Django являются удобным инструментом для получения данных от пользователя и сохранения их в базе данных, на сервере. Если ваш сайт не является личным блогом, а подразумевает взаимодействие с читателями, освоение форм будет обязательным пунктом в создании проекта.

Вывод

На этой лекции мы:

1. Узнали о формах Django
2. Разобрались в создании классов форм
3. Изучили поля и виджеты форм
4. Узнали о обработке данных из формы
5. Разобрались в сохранении файлов

Домашнее задание

1. Для закрепления материалов лекции попробуйте самостоятельно набрать и запустить демонстрируемые примеры.
2. *Загляните в официальную документацию Django и изучите дополнительные возможности работы с формами.