Django: формы | Я.Шпора

Формы в HTML

HTML-форма — это набор HTML-элементов, которые отрисовывают в браузере интерфейс для ввода информации. При отправке формы браузер отправляет на сервер запрос, передавая в нём данные, которые пользователь ввёл в форму. Адрес и тип запроса описываются в атрибутах тега `<form>`.

В самом простом варианте вёрстки HTML-код формы может выглядеть так:

```
...
<form>
  <input type="text" name="user_name"> <!-- Поле ввода -->
  <input type="submit" value="Отправить"> <!-- Кнопка для отправки форм
ы -->
</form>
...
```

`<form>` — парный тег, который определяет начало и конец формы в коде.

`<input>` — многофункциональный одиночный тег, который в зависимости от указанных атрибутов может отрисовываться по-разному, например — как поле для ввода текста, кнопка, переключатель, флажок.

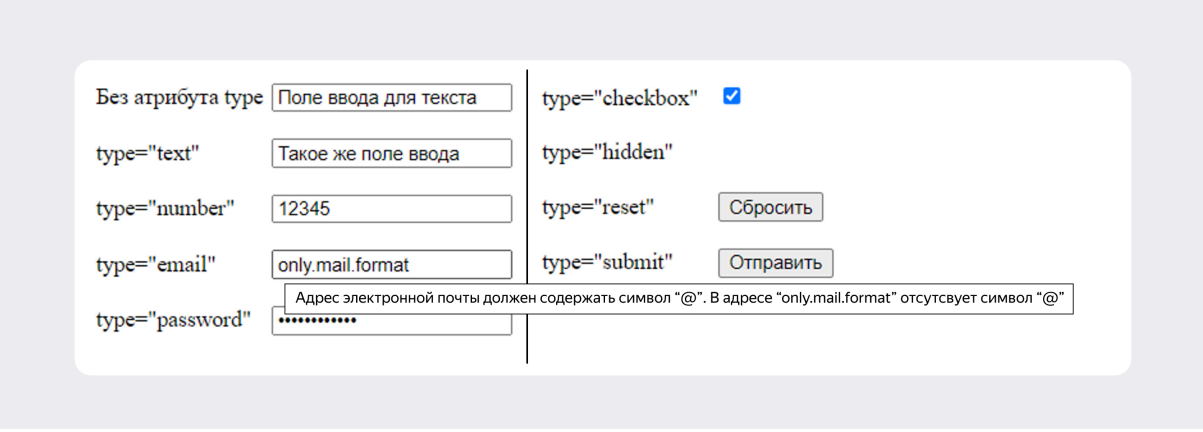
Атрибуты тега <input>

Атрибут `type` тега `<input>` сообщает браузеру, в каком виде отображать этот элемент.

Вот лишь некоторые значения, которые может принимать атрибут `type`:

Значение атрибута type	Тип отображаемого элемента
text	text — это значение атрибута type по умолчанию; если type не указан — input будет выглядеть именно как поле ввода
number	Поле для ввода чисел (попытка ввести другие символы ничего не даст)
email	Поле для ввода адреса электронной почты
password	Поле для ввода текста; при вводе символы заменяются точками или звёздочками
checkbox	Флажок («галочка»)
hidden	Скрытое от пользователя поле
reset	Кнопка для очистки формы
submit	Кнопка для отправки данных

Если в `type` указано, что поле должно принимать только специфическое содержимое (например `number` или `email`), то перед отправкой данных браузер проверит, соответствует ли введённая информация требованиям. Если в поле будут обнаружены неразрешённые значения — форма не будет отправлена, а пользователь получит сообщение о проблеме.



Остальные атрибуты тега `<input>`:

- Атрибут `name` — имя элемента; при отправке данных на сервер `name` преобразуется в ключ для значения, которое ввёл пользователь.
- Атрибут `required` делает поле обязательным для заполнения.

- Атрибут `value`: на кнопках `<input type="submit">` или `<input type="reset">` — определяет надпись на кнопке; для полей ввода — позволяет заранее заполнить их содержимым по умолчанию.

Заголовки полей, тег <label>

Для удобства пользователей к полям формы можно добавить название; это делается с помощью тега `<label>`.

```
<form>
  <label>Введите имя: </label>
  <input type="text" name="user_name" required>
  <input type="submit" value="Отправить">
</form>
```

Содержимое тега `<form>` можно форматировать с помощью любых HTML-тегов. Например, при помощи тега `<p></p>`:

```
<form>
  <p>
    <label>Введите имя: </label>
    <input type="text" name="user_name" required>
  </p>
  <p>
    <label for="is_human">Я человек</label>
    <input id="is_human" type="checkbox" name="human"><br>
    <small>Не ставьте галочку, если вы не человек</small>
  </p>
  <p><input type="submit" value="Отправить"></p>
</form>
```

Атрибуты тега <form>

Самые часто используемые атрибуты тега `<form>`:

`method` (тип запроса), например: `<form method="post">`;

`action` (адрес, куда будет отправлен запрос), например: `<form action="https://yandex.ru/search/">`.

Если эти атрибуты не указаны, то применяются настройки по умолчанию: отправляется GET-запрос на тот же адрес, где расположена страница с формой.

Формы в Django

Формы в Django генерируются из объектов классов, унаследованных от класса `Form` из пакета `django.forms`. Объект формы передаётся в шаблонизатор, а в шаблонизаторе генерируется HTML-код веб-формы.

Класс Forms

Чтобы создать форму в Django, нужно объявить класс-наследник от класса `Form` из пакета `django.forms`.

```
from django import forms


class BirthdayForm(forms.Form):
    first_name = forms.CharField(max_length=20)
    last_name = forms.CharField(required=False)
    birthday = forms.DateField()
```

Объект формы передаётся из view-функции в шаблон в словаре контекста под ключом `form` (это традиционное название, хотя назвать этот ключ можно как угодно):

```
# views.py
from django.shortcuts import render

# Импортируем класс BirthdayForm, чтобы создать экземпляр формы.
from .forms import BirthdayForm


def birthday(request):
    # Создаём объект класса формы.
    form = BirthdayForm()
    # Добавляем его в словарь контекста под ключом form:
    context = {'form': form}
```

```
    # Указываем нужный шаблон и передаём в него словарь контекста.
    return render(request, 'birthday/birthday.html', context)
```

Чтобы отобразить форму в шаблоне, используются двойные фигурные скобки, как и для вывода любой другой переменной в шаблоне:

```
<!-- templates/birthday/birthday.html -->
{% extends "base.html" %}

{% block content %}
  <form>
    {{ form }}
    <input type="submit" value="Submit">
  </form>
{% endblock %}
```

Django не генерирует тег `<form></form>`, его необходимо написать самостоятельно.

Подписи и подсказки к полям формы

По умолчанию заголовки полей HTML-формы генерируются из имён атрибутов класса формы: имя атрибута `first_name` превратится в лейбл *First name*, а `last_name` — в лейбл *Last name*.

Собственные названия для полей можно установить в аргументе `label`, а подсказку для поля — в аргументе `help_text` нужного поля.

```
...
class BirthdayForm(forms.Form):
    first_name = forms.CharField(label='Имя', max_length=20)
    last_name = forms.CharField(
        label='Фамилия', required=False, help_text='Необязательное поле'
    )
    birthday = forms.DateField(label='Дата рождения')
```

Типы полей ввода input

В HTML тип поля ввода устанавливается с помощью атрибута type тега `<input>`. В веб-форме Django настроить тип поля можно с помощью **виджета**, применённого к полю Django-формы. Виджеты импортируются из `django.forms`.

Например, поле для даты можно представить на странице в виде календаря:

```
# birthday/forms.py
from django import forms


...
class BirthdayForm(forms.Form):
    ...
    birthday = forms.DateField(
        label='Дата рождения',
        # Указываем, что виджет для ввода даты должен быть с типом date.
        widget=forms.DateInput(attrs={'type': 'date'})
    )
```

Виджеты применяются и для других типов отображения полей.

Доступ к данным, отправленным из веб-формы

Данные, отправленные из формы, становятся доступны в объекте запроса `request`, в атрибуте `request.GET` или `request.POST`, в зависимости от типа запроса.

Валидация полученных данных в объекте формы

Для валидации:

- данные из запроса (`request.GET` или `request.POST`) передают в объект формы:

```
form = BirthdayForm(request.GET)
```

- затем вызывают метод `form.is_valid()`: он возвращает `True`, если данные из запроса соответствуют типам данных, ожидаемым в форме.

```
def birthday(request):
    if request.GET:
        # Передаём параметры запроса в конструктор класса формы.
        form = BirthdayForm(request.GET)
        # Если данные валидны...
        if form.is_valid():
            # ...то выполняем необходимые действия.
            pass
    ...
```

При успешной валидации Django передаёт полученные в запросе значения полей в специальный словарь `form.cleaned_data`; ключи словаря совпадают с названиями полей, а значения приведены к нужным типам данных. Для дальнейшей работы значения надо брать из словаря `cleaned_data`.

Валидация полученных данных в шаблоне

Валидность формы можно проверять и в шаблоне. Для этого используется конструкция `{% if form.is_valid %}`:

```
{% with data=request.GET %}
  {% if form.is_valid %}
    <h2>Привет, {{ data.first_name }} {{ data.last_name}}</h2>
  {% endif %}
{% endwith %}
```

Формы на основе моделей

При создании формы на основе модели нужно

1. Создать класс формы, унаследованный от `forms.ModelForm`.
2. В подклассе `Meta` созданного класса указать модель, на основе которой должна быть построена форма.
3. Указать поля, которые пользователь должен увидеть в HTML-форме. В форме не обязательно использовать все поля модели. Определить список полей можно с помощью атрибутов `fields` или `exclude`:
   - `fields = ('first_name', 'birthday')` — в форме будут показаны только перечисленные поля модели;

- `exclude = ('last_name',)` — в форме будут показаны все поля модели, за исключением перечисленных;
- если в форме нужно использовать все поля модели, то указывается значение `fields = '__all__'`.

```
# birthday/forms.py
from django import forms

# Импортируем класс модели Birthday.
from .models import Birthday


# Для использования формы с моделями наследуемся от forms.ModelForm.
class BirthdayForm(forms.ModelForm):
    # Все настройки задаём в подклассе Meta.
    class Meta:
        # Указываем модель, на основе которой должна строиться форма.
        model = Birthday
        # Указываем, что надо отобразить все поля.
        fields = '__all__'
        # Указываем виджеты для полей:
                widgets = {
            'birthday': forms.DateInput(attrs={'type': 'date'})
        }
```

CSRF-токен

В Django есть встроенная защита от CSRF: она применяется к формам, запросы из которых изменяют состояние данных на сервере (например, для форм, из которых отправляются POST-, DELETE- и PATCH-запросы).

Защита основана на встраивании в форму CSRF-токена. Токен встраивается через добавление тега `{% csrf_token %}` в код формы в шаблоне.

```
...
<!-- Для форм, отправляющих POST-запросы, csrf-токен необходим! -->
<form method="post">
  {% csrf_token %}
  ...
```

```
</form>
...
```

Сохранение данных формы в БД

Встроенный метод `save()` класса `ModelForm` позволяет сохранить данные из формы в БД. После сохранения метод `save()` возвращает сохранённый объект.

```
...

def birthday(request):
    form = BirthdayForm(request.POST or None)
    context = {'form': form}
    if form.is_valid():
        form.save()
        ...
```

Редактирование объекта модели через форму

Для редактирования объекта через форму необходимо

- получить из POST-запроса `id` той записи, которую надо изменить,
- извлечь из базы данных объект с этим `id`,
- передать полученный объект в конструктор формы через параметр `instance`.

```
# Находим запрошенный объект для редактирования по первичному ключу
# или возвращаем 404 ошибку, если такого объекта нет.
instance = get_object_or_404(Birthday, pk=pk)
# Связываем форму с найденным объектом: передаём его в аргумент instance.
form = BirthdayForm(request.POST or None, instance=instance)
...
```

Кастомный валидатор формы

Кастомный валидатор данных — это функция, принимающая на вход данные из формы и выбрасывающая исключение, если валидация не пройдена:

```python
# <app_dir>/validators.py
# Импортируем класс для работы с датами.
from datetime import date

# Импортируем ошибку валидации.
from django.core.exceptions import ValidationError


# Пишем функцию-валидатор; на вход она будет принимать дату рождения
# и проверять, что значение age укладывается в пределы от 1 до 120.
def real_age(value: date) -> None:
    # Считаем разницу в днях между сегодняшним днём и днём рождения
    # и делим на 365.
    age = (date.today() - value).days / 365
    # Если возраст меньше 1 года или больше 120 лет — выбрасываем ошибку
    if age < 1 or age > 120:
        raise ValidationError(
            'Ожидается возраст от 1 года до 120 лет'
        )
```

В формах, наследуемых от `forms.Form`, валидатор подключается к классу формы: имя валидатора указывается в описании поля.

```python
# <имя_приложения>/forms.py
from django import forms

# Импортируем функцию-валидатор.
from .validators import real_age


class BirthdayForm(forms.Form):
    first_name = forms.CharField(label='Имя', max_length=20)
    last_name = forms.CharField(
        label='Фамилия', required=False, help_text='Необязательное поле'
    )
    birthday = forms.DateField(
        label='Дата рождения',
```

```
        widget=forms.DateInput(attrs={'type': 'date'}),
        # В аргументе validators указываем список или кортеж
        # валидаторов этого поля (валидаторов может быть несколько).
        validators=(real_age,),
    )
```

В формах, наследуемых от `forms.ModelForm`, валидатор подключается прямо к модели, с которой связана форма; имя валидатора указывается в описании поля модели.

```
from django.db import models

# Импортируется функция-валидатор.
from .validators import real_age


class Birthday(models.Model):
    first_name = models.CharField('Имя', max_length=20)
    last_name = models.CharField(
        'Фамилия', max_length=20, help_text='Необязательное поле', blank=
    )
    # Валидатор указывается в описании поля.
    birthday = models.DateField('Дата рождения', validators=(real_age,))
```

Метод clean для полей

Дополнительная валидация и очистка данных выполняется **clean-методами** полей формы. Эти методы ничего не принимают на вход: они автоматически берут значения из словаря `cleaned_data` (он доступен в экземпляре формы `self`).

Имена clean-методов составляются по принципу `clean_<имя_поля>`.

- Если clean-метод применяется для валидации — он вернёт то же значение, которое получили (если валидация прошла успешно).
- Если clean-метод применяется для очистки данных — он вернёт новое, «очищенное» значение.

Значение, которое возвращает clean-метод, записывается в словарь `cleaned_data`, заменяя прежнее значение.

```
...
class BirthdayForm(forms.ModelForm):

    class Meta:
        ...

    def clean_first_name(self):
        # Получаем значение имени из словаря очищенных данных.
        first_name = self.cleaned_data['first_name']
        # Разбиваем полученную строку по пробелам,
        # возвращаем только первую часть имени:
        # если имя состоит из нескольких слов - "очищаем" имя до одного с
        return first_name.split()[0]
```

Метод clean() для формы

Метод `clean()` применяется для валидации взаимозависимых полей.

```
from django import forms
# Импортируем класс ошибки валидации.
from django.core.exceptions import ValidationError

from .models import Birthday

# Проверим, что никто из пользователей не представится именем одного из б
# Множество с именами участников ливерпульской четвёрки.
BEATLES = {'Джон Леннон', 'Пол Маккартни', 'Джордж Харрисон', 'Ринго Стар


class BirthdayForm(forms.ModelForm):

    class Meta:
        ...

    def clean_first_name(self):
```

```
        ...

    def clean(self):
        # Получаем имя и фамилию из очищенных полей формы.
        first_name = self.cleaned_data['first_name']
        last_name = self.cleaned_data['last_name']
        # Проверяем, не совпадает ли комбинация имени и фамилии
        # с именами в множестве BEATLES.
        if f'{first_name} {last_name}' in BEATLES:
            raise ValidationError(
                'Мы тоже любим Битлз, но введите, пожалуйста, своё настоя
            )
```

При необходимости метод `clean()` может возвращать обновлённый словарь `cleaned_data`, но в листинге проводится только проверка данных, так что возвращать ничего не нужно.

Практикум