Л6. Обработка POST-запросов

Дата	
Лектор	
↗ Предыдущая лекция	Л5. Обработка GET-запросов с помощью Django ORM
↗ Следующая лекция	Л7. Работа с моделью пользователя

Л5. Обработка GET-запросов с помощью Django ORM

Л7. Работа с моделью пользователя

▼ Ссылки для добавления к лекции:

- `https://docs.djangoproject.com/en/3.2/topics/email/#module-django.core.mail` - Отправка писемв Django
- `https://docs.djangoproject.com/en/3.2/ref/forms/api/#module-django.forms` - Работа с Формами в Django
- `https://docs.djangoproject.com/en/3.2/ref/forms/widgets/#built-in-widgets` - Виджеты для отображения полей

▼ Получение данных от пользователя

▼ Различные типы запросов

На прошлой лекции мы с вами подробно разбирали GET запросы, они составляют основную массу запросов, которые происходят в интернете. Вообще почти всегда читающих запросов на порядки больше изменяющих. Это очень важных фактор, который используют при проектировании различных систем, ведь сохранить какой-нибудь файл на сервере сильно дороже и медленнее, чем прочитать его, но благо, как только что было сказано, это случается редко.

Помимо GET есть ещё один основной тип запросов POST. Его назначение - отправить на сервер какие-то данные. У этого запроса в отличии от GET есть “тело”, то есть клиент может что-то передать. Стоит

понимать, что это очень универсальный запрос. Разработчики HTTP протокола понимали, что тип запроса это некоторая формальность, просто текстовая строка, поэтому появились и другие типы: DELETE, HEAD, OPTION, PUT, PATCH, …. Вообще, вы можете придумывать и свои типы запросов, если вам они нужны будут для разработки вашего бекенда. Но важно понимать, что когда вы взаимодействуете со стандартными инструментами, то у вас не будет такого выбора. Например, по стандарту в браузерах есть только GET и POST. Да, можно эмулировать и другие с помощью JS, но это уже более сложные механики, которые мы с вами не будем обсуждать.

Наша с вами задача - познакомиться с тем, как из браузера отправить данные в Django стандартными путями. Если вы будете вбивать адреса в адресную строку, то браузер будет всегда отправлять GET запрос. Чтобы отправить POST в HTML есть формы. Это возможность получить от пользователя входные данные и отправить их на сервер. Отправлять их можно и GET запросами, в этом случае данные будут кодироваться в URL, у такого подхода есть свои минусы.

```
<form action="адрес обработчика" method="post" enctype="multipart/form-data">
</form>
```

Основу любой формы составляет элемент `<form>...</form>`. Он не предусматривает ввод данных, так как является контейнером, удерживая вместе все элементы управления формы – **поля**. Важный момент, что, по крайней мере для нас, обязательным полем будет поле с кнопкой отправки формы, оно не добавляется само, потому что триггером для отправки может быть множество вариантов и подлежит кастомизации.

`action` — необязательное поле, указывается, если обрабатывающий ендпоинт отличается от текущего.

`method` — метод запроса (может быть POST и GET), некоторые браузеры позволяют использовать и другие значения, но мы не знаем браузер всех наших клиентов. Так как мы планируем изменять данные на сервере, то следует использовать `POST`.

`enctype="multipart/form-data"` — указывается, если планируется передавать файлы, есть и другие значения, но если он вам понадобится, что если что-то не работает, вы сможете сами его найти. В целом этот параметр просто указывает, как кодируются и передаются данные. А с файлами

есть сразу очевидные сложности: они большие, а значит надо будет отправлять кусками, плюс в них может встречаться произвольная последовательность данных, что стоит учитывать, поэтому об этой ситуации рассказываем сразу.

```
<form method="post">
  <ul>
    <li>
      <label for="name">Имечко</label>
      <input type="text" id="name" name="name">
    </li>
    <li>
      <label for="mail">Почта</label>
      <input type="email" id="mail" name="mail">
    </li>
    <li class="button">
      <button type="submit">Отправить</button>
    </li>
  </ul>
</form>
```

Вот простая форма, современные браузеры при показе их будут проверять данные, подсказывать что-то. Например в поле mail можно вбить только почту. Важно понимать, что эта валидация происходит исключительно в браузере и её легко обмануть. Например, можно отредактировать DOM-дерево браузера или вообще отправить запрос из консоли. В этом случае вообще все проверки на клиентской стороне будут пропущены. Поэтому закон номер один - любые данные, полученные от пользователя, надо проверять! Было бы хорошо, если это делали бы не мы сами каждый раз, а можно было вызывать какие-то готовые компоненты, чтобы это всё отдать им на откуп. В Django как раз такие компоненты есть, сегодня мы с вами познакомимся с ними.

Показываем работу проверки в браузере, редактирование DOM дерева, копировать как curl, редактировать запрос и отправку из консоли. Пока можно слать запросы вникуда, не суть важно.

▼ Создание форм в Django

Давайте поместим написанную форму в шаблон и попробуем получить данные.

Добавляем, открываем, нажимаем отправить и видим ошибку

Не работает. Тут Django опять умнее нас и хочет обезопасить наш сайт от потенциальной известной атаки. Эта защита включена по умолчанию. В частности для этого и задавался `SECRET_KEY` в настройках. Ошибка весьма говорящая, что у нас не приложен токен. Сначала давайте поговорим про саму атаку.

Пару слов про атаку, чем опасна и как этот токен помогает защититься. Конечно, кража самого токена уже смертельна, но они очень короткоживующие.

> CSRF (англ. cross-site request forgery — «межсайтовая подделка запроса», также известна как XSRF) — вид атак на посетителей веб-сайтов, использующий недостатки протокола HTTP. Если жертва заходит на сайт, созданный злоумышленником, от её лица тайно отправляется запрос на другой сервер (например, на сервер платёжной системы), осуществляющий некую вредоносную операцию (например, перевод денег на счёт злоумышленника). Для осуществления данной атаки жертва должна быть аутентифицирована на том сервере, на который отправляется запрос, и этот запрос не должен требовать какого-либо подтверждения со стороны пользователя, которое не может быть проигнорировано или подделано атакующим скриптом.

Конечно, можно просто отключить проверку, но мы точно не будем так поступать. Чтобы починить это нужно проверить, что нужный middleware активен и мы приложили этот токен. Первое делается очень просто в настройках проекта.

```
MIDDLEWARE = [
    'django.middleware.csrf.CsrfViewMiddleware',
]
```

Второе исправляется не сложнее нужно добавить `{% csrf_token %}` в форму.

Стоит понимать, что в реальной жизни этот токен иногда передается не в поле формы, а в заголовках запроса, но это более хитрые сущности, иногда они нужны, например, такой подход повышает безопасность, ведь в этом случае токен не появляется в коде страницы, а живёт только в памяти браузера.

Теперь давайте проверим, что всё работает. Ну а чтобы убедиться в этом, мы можем просто вывести в консоль введенные пользователям данные. Они доступны через `request.POST`. Там хранится специальный объект `QueryDict`. В нашем случае в нём будут следующие ключи: `'csrfmiddlewaretoken'`, `'name'`, `'email'`.

```
if request.method == 'POST':
    logger.info('Post data: %r', request.POST)
    name = request.POST.get('name')
    mail = request.POST.get('mail')
    logger.info(  # Нужно сконфигурить логгер!
        'name: %s mail %s',
        request.POST.get('name'),
        request.POST.get('mail'),
    )
```

И так, данные мы получили, теперь надо научиться их валидировать. Как я уже говорил, в Django есть способ это всё автоматизировать, просто задать набор требований к полям и всё. Делается это аналогично уже знакомой нам механике моделей. Но если модели отвечают за работу с базой, то формы могут существовать без неё. Поэтому есть Django.Forms.

▼ Forms

Но идея делать форму, валидировать руками не очень. Лучше как-то автоматизировать процесс. Для этого в Джанго есть Form

Создаём

```
from django import forms


class FeedbackForm(forms.Form):
    name = forms.CharField(
        label='Имя',
        max_length=100,
        help_text='Текст для подсказки'
    )
    mail = forms.CharField(label='Почта', max_length=100)
```

Чтобы отрисовать эту форму, надо отдать её в шаблон через контекст, как мы делаем с остальными данными. Важное отличие, что мы передаем начальные значения формы, чтобы не мучать пользователя, каждый раз требуя вводить все значения, если что-то пошло не так. Как вы уже поняли, что основная идея, которую мы постоянно упоминаем, это счастье пользователя, ему должно быть удобно.

```
from .forms import FeedbackForm


def home(request):
    template = 'homepage/home.html'
    form = FeedbackForm(request.POST or None)
    context = {
        'form': form,
    }
    return render(request, template, context)
```

Форма имет возможность автоматической генерации. Но csrf_token и кнопку отправки надо сделать самому.

```
<form  method="post">
  {% csrf_token %}
  {{ form }}
  <input type="submit" value="Поехали!">
</form>
```

Есть и другие виды автогенерации форм в шаблонах, есть три стандартных методов отображения формы:

- `{{ form.as_table }}` выведет их в таблице, в ячейках тега `<tr>`
- `{{ form.as_p }}` обернет их в тег `<p>`
- `{{ form.as_ul }}` выведет в теге `<li>`

▼ Валдиация данных

Чтобы показать ошибку, можно подумать, что нужно вернуть ошибку валидации в запросе, но нет. Это надо делать в валидаторах формы, аналогично моделям. То есть такой код работать не будет

```
if request.method == 'POST':
    mail = request.POST.get('mail')
    if not 'ya' in mail:
        raise forms.ValidationError('У почты нет яндесовости')
```

Всё, что мы можем сделать уже в обработке запроса, это использовать данные из формы и что-то сделать, после чего, например, переадресовать пользователя на другую страницу. Конечно, мы можем делатиь и ещё много всего, если залезим в потроха формы, но мы ведь не хотим делать костыли.

У форм есть метод is_valid, который возвратит состояние валидации данных, чтобы спокойно можно было использовать информации из формы. Валидатор не роняет проект, а отдает в шаблон текст из ValidationError. Такие ошибки сохраняются в форму или как ошибки конкретного поля, либо как общие ошибки. Стандартная генерация HTML для формы умеет их отображать. Таким образмо код получается таким

```
item_form = ItemForm(request.POST or None)
if request.method == 'POST' and form.is_valid():
    ...
```

Формы в браузерах умеют поддерживать разнообразные валидации, например, можно попросить следить за тем, чтобы в форме был введён почтовый адрес или валидный URL. Но как мы выяснили ранее, давать это на откуп браузеру опасно. Поэтому нам надо это проверять и на backend. К счастью, такие стандартные вещи поддержаны в Django из коробки. А вот что-то нестандартное, например, проверка первой буквы, прийдется реализовывать самим.

```
def start_with_a(value):
    if value[0] != 'A':
        raise forms.ValidationError('Должно начинаться с A')
    return value


class FeedbackForm(forms.Form):
    name = forms.CharField(
        label='Имя',
        max_length=100,
        validators=[start_with_a]
    )
```

```
    email = forms.EmailField(
        label='Почта',
        max_length=100,
    )
```

Использование готовых компонентов позволит браузеру правильно показать тип поля, на мобильных устройствах показать правильную раскладку, ну а не бэкенде обязательно запустит валидацию.

Конечно, мы тут с вами говорим про простые поля. Если же требуется проверять загруженные файлы, например, или что-то ещё хитрое, то их обрабатывтаь нужно будет самим как-то отдельно. Для этого всегда есть доступ к `request.POST` и `request.FILES`. Но мы с вами этой темы коснёмся только слегка, без проверки типов. Проверка картинок за нас лежит в ImageField, а кроме картинок у нас будет ограничение просто на то, что это “файл”, то есть никаких.

▼ Форма из модели

Как уже говорилось, формы очень похожи на модели. Но кроме всего прочего, формы часто используются для работы с моделями, поэтому логично, что в Django можно создавать формы по моделям, для этого есть специальный базовый класс.

```
class Item(models.Model):
    name = models.CharField(
        'Название',
        max_length=150,
        help_text='Максимум 150 символов',
    )


class ItemForm(forms.ModelForm):
    class Meta:
        model = Item
        fields = (Item.name.field.name,)
```

Конечно, они должны быть разнесены по свои файлам `models.py` и `forms.py`.

Но как быть с дополнительными полями, которые дают нам формы? Например, подсказки. У моделей их не было. Django даёт возможность их доопределить через Meta класс

Переопределяем label и help_text.

```
class ItemForm(forms.ModelForm):
    class Meta:
        model = Item
        fields = (Item.name.field.name,)

        labels = {
            Item.name.field.name: 'Имя поля',
        }
        help_texts = {
            Item.name.field.name: 'Подсказка',
        }
```

В этом случае порядок вызова валидаторов определён следующим образом: на уровне модели валидация начинается строго после того, как отработали все проверки на уровне формы.

▼ CRUD

Продолжая аналогию с базой, запросы на сервер тоже могут быть разделены на 4 типа CRUD. Стандартные средства браузера не позволяют нам разнести их по разным методам, только с помощью JS, хотя такие методы есть в стандарте HTML. Поэтому стандартный паттерн в этом случае - создать несколько ендпоинтов с говорящими названиями и каждый обрабатывать. Главное, есто если это запрос на удаление, то всегда проверять, что он был сделан POST, такие запросы не перезапрашиваются браузером автоматом, а значит это некоторая защита. Сами URL обычно имеют вид:

- Create `item/`
- Read `item/pk/`
- Update `item/pk/update/`
- Delete `item/pk/delete`

```
from django.shortcuts import redirect, render

from .forms import ItemForm
from .models import Item


def item_create(request):
    template = 'create.html'
    form = ItemForm(request.POST or None)
    if form.is_valid():
        name = form.cleaned_data['name']
```

```
        item = Item.objects.create(
            name=name
        )
    context = {'form': form}
    return render(request, template, context)


def item_delete(request, pk):
    template = 'delete.html'
    item = get_object_or_404(Tag, pk=pk)
    if request.method == 'POST':
        item.delete()
    return render(request, template, {'item': item})


def item_update(request, pk):
    template = 'update.html'
    item = get_object_or_404(Tag, pk=pk)
    form = ItenForm(request.POST or None)
    if form.is_valid():
        item.name = form.cleaned_data['name']
        item.save(update_fields=['name'])
    context = {'form': form}
    return render(request, template, context)
```

▼ Работа с полученными данными

С созданием форм разобрались. Давайте теперь обработаем информацию. Когда в форму добавлены данные, и она проверена методом `is_valid()` (и `is_valid()` вернул `True`), проверенные данные будут добавлены в словарь `form.cleaned_data`. Эти данные будет преобразованы в подходящий тип Python. То есть мы можем быть уверены, что там лежит целое число или почтовый адрес, больше об этом думать не надо. И с обработанными данными можно работать. Например, отправлять письмо или сохранять в БД

▼ Эмуляция почтового сервера и сохранение писем в файлы

Самой распостраннёй реакцией сервера на форму бывает отправка письма. Недостаточно просто показать пользователю, что всё хорошо, хочется оставить им запись в “логе”, чтобы пользователь мог всегда вернуться к ней. Почта тут очень хорошо подходит. И это может быть и создание заказа, отправка обратной связи, …

В Django есть несколько модулей для отправки писем, подключить их можно через ключ конфигурации EMAIL_BACKEND в `settings.py`:

```
EMAIL_BACKEND = 'django.core.mail.backends.<название модуля>'
```

Встроенные в Django почтовые модули могут отправлять почту по протоколу SMTP, выводить в консоль содержимое письма, сохранять письма в файлы, хранить их в памяти или просто ничего не делать (даже для «ничего не делать» написан специальный модуль).

Что бы не бороться с особенностями настройки сервисов, а сразу присутпить к делу, можно настроить **эмуляцию** работы почтового сервера. Подключите к проекту модуль **filebased.EmailBackend**: он будет сохранять текст отправленных электронных писем в файлы в отдельную директорию. Ведь подключать настоящую почту сильно сложнее и имеет свои требования.

Для этого добавьте в `settings.py` такой код:

```
# Подключаем движок filebased.EmailBackend:
EMAIL_BACKEND = 'django.core.mail.backends.filebased.EmailBackend'
# Указываем директорию, в которую будут складываться файлы писем:
EMAIL_FILE_PATH = BASE_DIR / 'sent_emails'
```

Теперь при отправлении письма система будет делать вид, что отправила письмо, но эти письма будут «отправляться» в директорию *sent_emails/*. Если эта директория не существует, она будет создана после первого «письма». Для отправки писем в Django существует несколько методов.

Самый простой из них — работа через `send_mail`. Вот как он описан в документации Django.

```
from django.core.mail import send_mail


send_mail(
    'Subject here',
    'Here is the message.',
    'from@example.com',
    ['to@example.com'],
    fail_silently=False,
)
```

В `send_mail` требуются передать обязательные параметры `subject`, `message`, `from_email` и `recipient_list`.

- `subject`: Тема письма (строка).
- `message`: Текст сообщения (строка).
- `from_email`: Если `None`, Django будет использовать значение параметра `DEFAULT_FROM_EMAIL`.
- `ecipient_list`: Получатели писем в ввиде списка строк. Каждый получатель из списка будет видеть других в поле «Кому:» электронного сообщения.

`fail_silently`: необязательный аргумент. Если установлен в значение False, при неудачной попытке отправки письма будет вызвано одно из исключений `smtplib.SMTPException`. Список возможных исключений — в документации `smtplib`, все они являются подклассами `SMTPException`.

Добавим в сообщение информацию из формы, а остальные парметры оставим такими же, как в документации Django.

```
from django.core.mail import send_mail
from django.shortcuts import redirect, render

from .forms import ItemForm


def feedback(request):
    template = 'template.html'
    form = ItemForm(request.POST or None)

    if form.is_valid():
        name = form.cleaned_data.get('name')
        send_mail(
            'Subject here',
            name,
            'from@example.com',
            ['to@example.com'],
            fail_silently=False,
        )

        return redirect('about:thanks')
    context = {
        'form': form,
    }
    return render(request, template, context)
```

Отправляем письмо видим его в *sent_emails/*

После обработки формы часто нужно перейти на страницу (например страницу подтверждения, что запрос обработан) Вот как это делается с помощью шотката:

```
def my_view(request):
    # return redirect('<name_space>:<view_name>', *args)
    return redirect('catalog:item_detail', tag.pk)
```

Конечно, результатом обработки формы могут быть любые действия, в нашем примере почта - как самый простой вариант.

▼ Тестирование

Ну и не можем не поговорить про тестирование. Пускай у нас есть форма из модели, надо проверить, что она может создавать объекты в базе, например. По сути главное уметь из тестов отправлять валидную форму, а как проверять реакцию - уже задача программиста, который писал сценарий обработки.

Если у нас есть модель, форма

```
class Item(models.Model):
    name = models.CharField(
        'Название',
        max_length=150,
        help_text='Максимум 150 символов'
    )


class ItemForm(forms.ModelForm):
    class Meta:
        model = Item
        fields = ('name',)
        labels = {
            'name': 'Имя',
        }
        help_texts = {
            'name': 'Подсказка',
        }
```

И view-функция

```


def item_create(request):
    template = 'template.html'
    form = ItemForm(request.POST or None)
```

```
    if form.is_valid():
        name = form.cleaned_data['name']
        item = Item.objects.create(
            name=name
        )
        return redirect('about:thanks')
    context = {'form': form}
    return render(request, template, context)
```

То мы можем написать следующий тест.

```
from django.test import Client, TestCase

from .forms import ItemForm
from .models import Item


class FormTests(TestCase):
    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        cls.form = ImaginForm()

    def test_name_label(self):
        name_label = FormTests.form.fields['name'].label
        self.assertEquals(title_label, 'Имя')

    def test_name_help_text(self):
        name_help_text = FormTests.form.fields['name'].help_text
        self.assertEquals(title_help_text, 'Подсказка')

    def test_create_task(self):
        """Валидная форма создает Item."""
        # Подсчитаем количество записей в Item
        items_count = Item.objects.count()
        form_data = {
            'name': 'Тест',
        }
        # Отправляем POST-запрос
        response = Client().post(
            reverse('app:endpoint'),
            data=form_data,
            follow=True
        )
        # Проверяем, сработал ли редирект
        self.assertRedirects(response, reverse('about:thanks'))
        # Проверяем, увеличилось ли число постов
        self.assertEqual(Item.objects.count(), itemss_count+1)
        # Проверяем, что создалась запись с заданным слагом
        self.assertTrue(
            Task.objects.filter(
                 name='Тест',
                ).exists()
        )
```

А то, как тестить, что контекст для пороверки form вы знаете из прошлого урока.

▼ Поддержка в HTML

Нам осталось немного поговорить про то, как можно отойти от стандартного отображения форм, ведь не всегда нам будет это подходить, автогенерация не всегда удобна.

▼ Цикл по полям и бутсьрап

На деле под капотом автогенерации используется примерно такой код. По форме можно итерироваться, получая последовательно поля формы.

```
{% for field in form %}
  <div>
    {{ field.label_tag }} {{ field }}
    {% if field.help_text %}
      <p class="help">{{ field.help_text|safe }}</p>
    {% endif %}
  </div>
{% endfor %}
```

Отображение ошибок при заполнении форм

Если происходит ошибка в заполнений форм, она сохраняется в свойство `errors`.

Есть два типа ошибок:

- не относящиеся конкретному полю (и/или ошибки скрытого поля), которые
  отображаются с помощью `{{ form.non_field_errors }}`
- И ошибки валидации поля `{{ form.<имя поля>.errors }}`, в которой хранится список ошибок заполнения поля.

Вот пример кода, который выводит форму включая все ошибки:

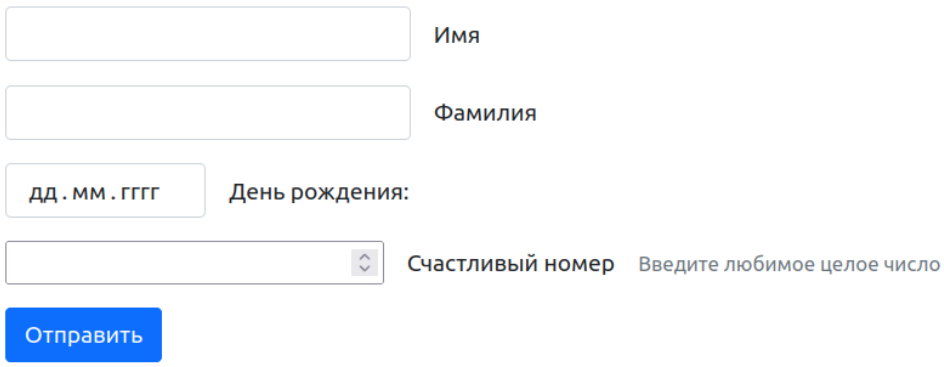
```
{% if form.errors %}
  {% for field in form %}
    {% for error in field.errors %}
      <div class="alert alert-danger">
        {{ error|escape }}
      </div>
```

```
    {% endfor %}
    {{ field.label_tag }} {{ field }}
    {% if field.help_text %}
      <p class="help">{{ field.help_text|safe }}</p>
    {% endif %}
  {% endfor %}
  {% for error in form.non_field_errors %}
    <div class="alert alert-danger">
      {{ error|escape }}
    </div>
  {% endfor %}
{% endif %}
```

Подробнее об ошибках, стилизации и работе с атрибутами формы в шаблонах смотрите в API форм.

Более сложные поля ввода

Если нам нужно будет ещё улучшить отображения, например, сделать красивые рамки полей, то нам нужно будет написать ещё больше HTML. А если учесть, что основная структура правильная и проблема только на полях input, то звучит сомнительно, что нам надо было бы всё это писать. Возьмем форму посложнее со стилями из примера бутстрепа.



Чтобы создать такую форму при помощи Django нужно перебрать подобную конструкцию в цикле.

```
<form method="post">
  {% csrf_token %}
  {% for field in form %}
    <div class="mb-3">
      <div class="row g-3 align-items-center">
        <div class="col-auto">
          {{ field }}
```

```
        </div>
        <div class="col-auto">
          {{ field.label_tag }}
        </div>
        <div class="col-auto">
          {% if field.help_text %}
            <span class="form-text">
              {{ field.help_text }}
            </span>
          {% endif %}
        </div>
      </div>
    </div>
  {% endfor %}
  <button type="submit" class="btn btn-primary">Отправить</button>
</form>
```

Этот код будет работать почти хорошо, но есть две проблемы:

- нужно добавить к каждому тегу `<input>` класс `form-control`.
- Некоторые поля (например, поле для вводы даты) Django рендерит, как `type="text"` . А это не дата. Неудобно.

▼ Виджеты

Писать все такие условия самим всегда в HTML - ужасно. И так написали много HTML. Но Django нас тут выручает. Для правильной отрисовки есть Виджеты. С их помощью можно добавлять классы и менять типы ввода.

В зависимости от того, создана форма при помощи Form или ModelForm синтаксис добавления виджетов к полям формы будет разным.

Для Form

В этом случае у нас есть полный контроль над полями, а значит можно явным образом описать прямо в полях формы. Из документации https://docs.djangoproject.com/en/3.2/ref/forms/widgets/#built-in-widgets выбирается подходящий по типу виджет и в него добавляются атрибуты `type` и `class`:

```
class PersonForm(forms.Form):
    name = forms.CharField(
        widget=forms.TextInput(attrs={'class': 'form-control'})
    )
```

```
    surname = forms.CharField(
        required=False,
        widget=forms.TextInput(attrs={'class': 'form-control'})
    )
    birth_date = forms.DateField(
        required=False,
        widget=forms.DateInput(attrs={'type': 'date', 'class': 'form-contro
l'})
    )
    lucky_number = forms.IntegerField(
        required=False,
        widget=forms.NumberInput(attrs={'class': 'form-control'})
    )
```

Форма примет нужный вид.

Для ModelForm

Если форма создана на основе модели, Django автоматически подберёт подходящие виджеты формы. Причем это будет работать и для связанных полей, на случай плохого выбора их можно переопределить, как и labels или help_text

```
class PersonForm(forms.ModelForm):

    class Meta:
        model = Person
        fields = '__all__'
        labels = {
            'name': 'Имя',
            'surname': 'Фамилия',
            'birth_date': 'Дата рождения',
            'lucky_number': 'Счастливый номер',
        }
        help_texts = {
            'lucky_number': 'Введите счастливый номер',
        }
        widgets = {
            'name': forms.TextInput(attrs={'class': 'form-control'}),
            'surname': forms.TextInput(attrs={'class': 'form-control'}),
            'birth_date': forms.DateInput(
                attrs={'type': 'date', 'class': 'form-control'}
            ),
            'lucky_number': forms.NumberInput(attrs={'class': 'form-contro
l'}),
        }
```

Но кажется странным, что надо перебирать все поля, неудобно. Есть и более элегантное решение переопределить базовые атрибуты

`forms.ModelForm` видимых полей в цикле, а в виджетах уже добавить только специфические атрибуты для отдельных полей.

```
class PersonForm(forms.ModelForm):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for field in self.visible_fields():
            field.field.widget.attrs['class'] = 'form-control'

    class Meta:
        model = Person
        fields = '__all__'
        labels = {
            'name': 'Имя',
            'surname': 'Фамилия',
            'birth_date': 'Дата рождения',
            'lucky_number': 'Счастливый номер',
        }
        help_texts = {
            'lucky_number': 'Введитье счастливый номер',
        }
        widgets = {
            'birth_date': forms.DateInput(attrs={'type': 'date'}),
        }
```

- если в модели есть поле типа *ForeignKey*, то в форме будет отрисовано поле выбора *ModelChoiceField*, `<select>` для выбора объекта связанной модели;
- для поля модели *ManyToManyField* в форме будет применён виджет *ModelMultipleChoiceField*, поле для множественного выбора из списка;

Таким образом, всё будет работать штатным образом и должно получаться красиво при автоматической генерации формы без нашего ведома. Конечно, если вы хотите более сложное что-то, то надо будет или ещё кастомизировать поля формы или писать генерацию форм самому.

▼ ДЗ

▼ Базовое

Создайте и зарегистрируйте приложение `feedback`, которое будет отвечать за обработку обратной связи. Для того, чтобы можно было

отправлять письмо пользователю, требуется в форме обратной связи принимать почту. Почта, с которой уходит письмо должна задаваться в env.

В нем создайте модель Feedback со следующими полями:

- text (текстовое поле)
- created_on (дата и время создания). Текущее серверное время и дата должны добавляться автоматически при создании записи.
- Почта

На основе модели создайте форму обратной связи.

Создайте адрес feedback/, обрабатываемый, view-функцией feedback, в шаблоне которого должна быть показана эта форма, сверстанная на бутстрап.

Доработайте главное меню, что бы в нем была ссылка на адрес со страницей формы (feedback:feedback).

После заполнения формы и отправки запроса, в папке send_mail/ должна появляться запись. Содержимое письма должно получаться из text. Остальные поля могут быть произвольными.

После заполнения формы создавался редирект на текущую страницу, где пользователю сообщается, что форма успешно отправлена.

Напишите тесты, проверяющие

- Форма предана в контекст
- Форма создается с правильными label и help_text
- Что после заполнения формы происходит редирект на страницу с формой

▼ Допник

Доработайте view-функцию, так что бы, что бы информация из формы не только отправлялась на почту, но и сохранялась в БД.

Напишите тесты, проверяющие, что после отправки валидной формы создаётся запись в БД

К модели обратной связи надо добавить статус обработки: “получено”, “в обработке” и “ответ дан”.

▼ Бонусная

Добавить в форму возможность загружать нескольких файлов в одном поле, которые сохраняются на сервере по пути `uploads/FEEDBACK_ID/...`.

Написать тест, который проверяет возможность загрузить файл.

Разделите модель так, что почта и другие персональные данные (если были добавлены) сохраняются в другой таблице в БД, отдельно от текста обращения.

▼ Ревьюерам, на что обратить внимание дополнительно.

1. Дата создания не добавляется в форму
2. Обработка формы происходит только по POST
3. В случае неудачи форма предзаполняется уже введёнными пользователям данные
4. Поля проверяются на корректность
5. Статус сделан через Choices
6. Поле для файлов помечено multiple
7. Почта пользователя/ФИО/номер телефона - персональные данные