

## ЛАБОРАТОРНАЯ РАБОТА № 10

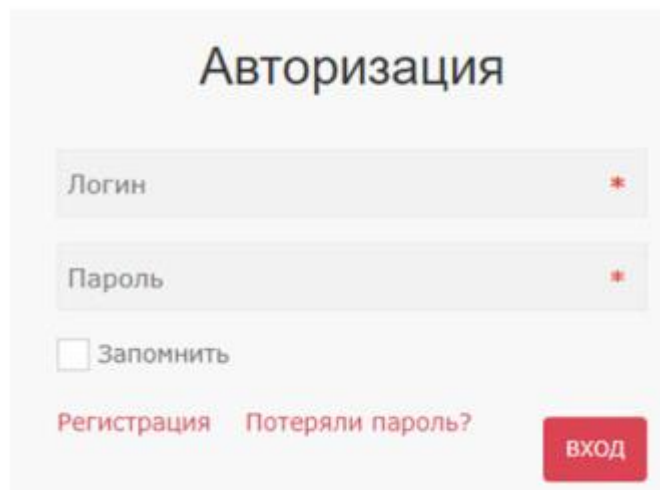
### Работа с формами

**Цель:** изучить работу с формами.

**Содержание отчета:** титульный лист, цель работы, задание, описание хода выполнения задания со скриншотами, листинги, вывод.

#### HTML-формы. Отправка данных по GET и POST-запросам

Формы – это один из важнейших элементов большинства сайтов. Например, когда мы выполняем авторизацию или регистрацию, то появляется страница с полями ввода, чекбоксами, кнопками, списками и другими элементами интерфейса:



Это и есть формы. В HTML они задаются тегом `<form>` и служат для передачи на сервер пользовательской информации, например, логина и пароля для входа на сайт. Посмотрим, как реализуются формы во фреймворке Django. Подробную документацию об этом можно почитать в разделе «Формы» на странице сайта:

<https://docs.djangoproject.com/en/4.2/#forms>

Формы в Django можно создавать в связке с моделью какой-либо таблицы БД. Тогда получаем формы, связанные с моделью. Например, когда мы выполняем авторизацию или регистрацию на сайте, то этот процесс, очевидно, связан с данными таблиц. Соответственно, используются формы, связанные с моделями. Либо можно создавать независимые формы, непривязанные к моделям. Например, когда создается простой поиск, или идет отправка письма на электронную почту. Если при этом обращение к БД не требуется, то и форма создается как независимая, самостоятельная.

Сначала мы рассмотрим форму, несвязанную с моделью, хотя и сделаем это на примере добавления статей в БД. Но, затем, модифицируем ее и превратим в форму, связанную с моделью.

Начнем с того, что в главном меню у нас уже есть пункт для добавления статей и функция представления **addpage** (в файле **women/views.py**). Изменим эту функцию так, чтобы она отображала шаблон **addpage.html**:

```
def addpage(request):  
    return render(request, 'women/addpage.html',  
        {'menu': menu, 'title': 'Добавление статьи'})
```

А сам шаблон **addpage.html** определим так:

```
{% extends 'base.html' %}  
  
{% block content %}  
<h1>{{title}}</h1>  
Содержимое страницы  
{% endblock %}
```

Теперь, при обновлении увидим полноценную страницу для добавления нового поста.

Давайте сначала сформируем форму вручную. Запишем вместо тега **<p>** и строки тег **form** в виде:

```
<form action=""></form>
```

Внутри этого тега для примера запишем следующий тег **input**:

```
<form action="">  
    <input type="text">  
</form>
```

Обновим страницу и увидим в окне браузера на странице поле ввода. Оно текстовое, так как имеет атрибут **type="text"**.

Давайте добавим еще несколько полей, например:

```
<form action="">  
    <input type="text">  
    <input type="checkbox">  
    <input type="number">  
    <input type="password">  
</form>
```

Типов этих полей много. Все их можно посмотреть в справочнике или на специализированных сайтах, например, по этой ссылке:

[https://www.w3schools.com/html/html\\_forms.asp](https://www.w3schools.com/html/html_forms.asp)

Чтобы добавить пояснения к этим полям, обычно, используют тег **label** следующим образом:

```
<form action="">
  <label for="id_1">Текст: </label><input type="text"
id="id_1">
  <label for="id_2">Отметка: </label><input
type="checkbox" id="id_2">
  <label for="id_3">Число: </label><input
type="number" id="id_3">
  <label for="id_4">Пароль: </label><input
type="password" id="id_4">
</form>
```

И расположим их в столбец, каждый на своей строке. Сделаем это с помощью тега абзаца **p**:

```
<form action="">
  <label for="id_1">Текст: </label><input type="text"
id="id_1"></p>
  <label for="id_2">Отметка: </label><input
type="checkbox" id="id_2"></p>
  <label for="id_3">Число: </label><input
type="number" id="id_3"></p>
  <label for="id_4">Пароль: </label><input
type="password" id="id_4"></p>
</form>
```

Пока форма не функциональна. Чтобы она отправляла данные на сервер, необходимо добавить кнопку специального типа **submit**:

```
<form action="">
  ...
  <button type="submit">Отправить</button></p>
</form>
```

Параметр **type** со значением **submit** внутри формы у кнопки можно и не прописывать, но так понятнее ее назначение.

Перейдем в форму, нажмем на эту кнопку и наша страница обновилась. Кроме того, в URL адресе появился знак вопроса. Он символизирует начало

строки с данными GET-запроса. Но почему сейчас никаких данных нет? Дело в том, что у каждого поля дополнительно нужно прописать специальный атрибут **name** с именем параметра поля. Сделаем это:

```
name="field_text"
name="field_check"
name="field_num"
name="field_psw"
```

Теперь при отправке формы GET-запрос стал принимать вид:

[http://127.0.0.1:8000/addpage/?field\\_text=&field\\_num=&field\\_psw=](http://127.0.0.1:8000/addpage/?field_text=&field_num=&field_psw=)

После знака вопроса через амперсанд идут данные в формате ключ=значение. Так как значений нет, то просто перечисляются поля.

Давайте введем какие-либо данные в форму и снова отправим их. Получим следующий вид GET-запроса:

[http://127.0.0.1:8000/addpage/?field\\_text=aaa&field\\_check=on&field\\_num=123&field\\_psw=asd](http://127.0.0.1:8000/addpage/?field_text=aaa&field_check=on&field_num=123&field_psw=asd)

Эти данные сервер передает во фреймворк Django, он их упаковывает в словарь **request.GET**, и там мы получаем к ним доступ. Причем данные отправляются в функцию **addpage()** по той причине, что мы в атрибуте **action** тега **form** ничего не указали. Если, например, мы там укажем слеш:

```
<form action="/">
...
</form>
```

то обработчиком формы будет считаться главная страница. Но нам это не нужно, оставим пустые кавычки.

У тега **form** есть еще один важный атрибут **method**, который определяет тип запроса для отправки данных формы. По умолчанию используется метод **GET**, то есть GET-запрос. Но этот запрос плох тем, что все данные пользователи видят в строке браузера. Например, так отправлять логины, пароли и другую служебную информацию не стоит. Вместо этого лучше воспользоваться другим типом запроса, который называется **POST**:

```
<form action="" method="post">
...
</form>
```

Но, если мы сейчас попробуем отправить данные, то возникнет ошибка из-за отсутствия специального поля **CSRF**. Оно нужно для защиты от межсайтовых атак, и фреймворк Django в целях безопасности его требует при POST-запросах. Добавим его. Для этого в начале формы следует прописать специальный шаблонный тег:

```
{% csrf_token %}
```

Теперь отправка данных работает без проблем, и в словаре **request.POST** мы можем все их видеть.

### Использование форм, несвязанных с моделями

Создадим нашу первую форму. Для этого в Django существует специальный класс **Form**, на базе которого удобно создавать формы, несвязанные с моделями:

<https://docs.djangoproject.com/en/4.2/ref/forms/api/#django.forms.Form>

Где следует объявлять формы? Обычно для этого создают в приложении отдельный файл **forms.py**. Мы так и сделаем (создаем файл **women/forms.py**). И в этом файле импортируем пакет **forms** и наши модели:

```
from django import forms
from .models import Category, Husband
```

Следующий шаг – объявить класс **AddPostForm**, описывающий форму добавления статьи. Он будет унаследован от базового класса **Form** и иметь следующий вид:

```
class AddPostForm(forms.Form):
    title = forms.CharField(max_length=255)
    slug = forms.SlugField(max_length=255)
    content = forms.CharField(widget=forms.Textarea())
    is_published = forms.BooleanField()
    cat =
forms.ModelChoiceField(queryset=Category.objects.all())
    husband =
forms.ModelChoiceField(queryset=Husband.objects.all())
```

Мы здесь определяем только те поля, с которыми будет взаимодействовать пользователь. Например, поля модели **time\_create** или **time\_update** нигде не фигурируют, так как заполняются автоматически. Каждый атрибут формы лучше назвать так же, как называются поля в таблице **women**. Впоследствии нам это облегчит написание кода.

В классе формы каждый атрибут – это ссылка на тот или иной экземпляр класса из пакета **forms**. Например, **title** определен через класс **CharField**, поле **is\_published** – через **BooleanField**, а список категорий **cat** – через класс **ModelChoiceField**, который формирует выпадающий список из данных, прочитанных из таблицы **Category**.

Какие классы существуют для формирования полей формы? Полный их список и назначения можно посмотреть на следующей странице документации:

<https://docs.djangoproject.com/en/4.2/ref/forms/fields/>

В частности, класс **CharField** служит для создания обычного текстового поля ввода, класс **BooleanField** – для checkbox'а, класс **ModelChoiceField** – списка с данными из указанной модели.

### Отображение формы в шаблоне

После того, как форма определена, ее можно использовать в функции представления **addpage()**. В самом простом варианте можно записать так:

```
def addpage(request):
    form = AddPostForm()
    return render(request, 'women/addpage.html',
{'menu': menu, 'title': 'Добавление статьи', 'form':
form})
```

Здесь создается экземпляр формы и через переменную **form** передается шаблону **addpage.html**. Осталось отобразить форму в нашем шаблоне. Перейдем в файл **addpage.html** и пропишем там следующие строки:

```
<form action="" method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Добавить</button>
</form>
```

Здесь в теге **<form>** через атрибут **action** в качестве обработчика указана текущая страница (пустые кавычки). Атрибут **method** определяет способ передачи информации на сервер (используется POST-запрос). В этом случае внутри формы обязательно записываем специальный тег **csrf\_token**, который генерирует скрытое поле с уникальным токеном. Это необходимо для защиты от CSRF-атак, когда вредоносный сайт пытается отправить данные от имени авторизованного пользователя. Фреймворк Django не станет обрабатывать данные, если отсутствует или не совпадает csrf-токен и, тем самым, защищает пользователя от подобных атак.

Следующая строка `{{ form.as_p }}` вызывает метод **as\_p** объекта формы для отображения ее полей с тегами абзацев `<p>`. Существуют и другие методы, которые формируют поля в виде элементов списка `<ul>` или в виде таблицы. Последний вариант, хоть и возможен, но считается устаревшей практикой. Здесь также стоит иметь в виду, что по умолчанию все поля в Django обязательны, если не указано обратное через параметр `required=False`.

Наконец, последняя строка – тег `<button>` создает кнопку типа **submit** для отправки данных формы на сервер и, в конечном итоге, нашей функции представления **addpage()**.

Если теперь обновить страницу, то увидим все указанные поля формы со списком и кнопкой. Заполним их, оставив флажок **is\_published** выключенным, и попробуем отправить данные. Браузер скажет нам, что поле **is\_published** является обязательным. Это не то поведение, которое нам нужно. Давайте все необязательные поля отметим, как необязательные. Для этого в классе **AddPostForm** у таких атрибутов пропишем параметр `required=False`:

```
class AddPostForm(forms.Form):
    title = forms.CharField(max_length=255)
    slug = forms.SlugField(max_length=255)
    content = forms.CharField(widget=forms.Textarea(),
required=False)
    is_published = forms.BooleanField(required=False)
    cat =
forms.ModelChoiceField(queryset=Category.objects.all())
    husband =
forms.ModelChoiceField(queryset=Husband.objects.all(),
required=False)
```

Теперь у нас три обязательных поля: **title**, **slug** и **cat**, остальные можно оставлять пустыми.

### Обработка данных формы

Форма в простейшем варианте у нас готова. Конечно, она пока выглядит просто, зато обладает нужным функционалом. Посмотрим, как можно обработать данные этой формы на стороне сервера, то есть в функции представления **addpage()**.

Определим функцию **addpage()** следующим образом:

```
def addpage(request):
    if request.method == 'POST':
        form = AddPostForm(request.POST)
        if form.is_valid():
```



```

        print(form.cleaned_data)
    else:
        form = AddPostForm()

    return render(request, 'women/addpage.html',
{'menu': menu, 'title': 'Добавление статьи', 'form':
form})

```

Как это работает? Сначала приходит обычный GET-запрос от браузера для открытия страницы по маршруту:

<http://127.0.0.1:8000/addpage/>

Условие не срабатывает и создается форма с пустыми полями, которая передается в шаблон и он, затем, отображается в браузере. Именно поэтому мы видим форму без данных. Далее, пользователь заполняет ее поля и отправляет на сервер по POST-запросу. Снова срабатывает наша функция представления **addpage()**, и на этот раз проверка проходит. Формируется объект формы с переданными данными и вызывается метод формы **is\_valid()**, который проверяет поля на корректность заполнения. Если проверка прошла, то в консоли отобразится словарь **form.cleaned\_data** полученных данных от пользователя. Если же проверка не пройдет, то пользователь увидит сообщение об ошибке.

Запустим веб-сервер, заполним нашу форму и нажмем на кнопку «Добавить». Если все заполнено корректно, то метод **is\_valid()** вернет **True** и в консоли отобразятся, так называемые, очищенные данные (**cleaned\_data**). При этом страница перезагрузится, но введенные данные не пропадут, так как мы в шаблон передаем объект **form** с заполненными полями, а не пустую.

Если же заполнить форму с ошибками, например в поле **slug** указать русские символы, то фреймворк Django автоматически сформирует сообщение об ошибке, что в поле **slug** присутствуют недопустимые символы. Соответственно, метод **is\_valid()** в этом случае вернет **False** и коллекция **cleaned\_data** не будет отображена в консоли. Нам даже не нужно делать обработку типовых ошибок. Фреймворк Django все берет на себя.

### Отображение полей формы

Сделаем улучшение внешнего вида формы добавления статей. Сейчас названия полей отображаются по-английски, и нам бы хотелось это изменить. Для этого у каждого класса поля формы есть специальный атрибут **label**, который позволяет задавать свои имена, например, так:

```

class AddPostForm(forms.Form):
    title = forms.CharField(max_length=255,
label="Заголовок")

```



```
slug = forms.SlugField(max_length=255, label="URL")
content = forms.CharField(widget=forms.Textarea(),
required=False, label="Контент")
is_published = forms.BooleanField(required=False,
label="Статус")
cat =
forms.ModelChoiceField(queryset=Category.objects.all(),
label="Категории")
husband =
forms.ModelChoiceField(queryset=Husband.objects.all(),
required=False, label="Муж")
```

Давайте для примера сделаем поле **is\_published** с установленной галочкой. Пропишем в классе **BooleanField** параметр `initial=True`. Также в классах **ModelChoiceField** добавим параметры `empty_label="Категория не выбрана"` и `empty_label="Не замужем"`, чтобы вместо черточек отображались по умолчанию в списке эти фразы.

Со всеми возможными параметрами можно ознакомиться в документации по ссылке:

<https://docs.djangoproject.com/en/4.2/ref/forms/fields/>

### Способы отображения формы в шаблонах

Посмотрим, что же в действительности представляет собой объект **form** внутри шаблона на примере ручного перебора и отображения всех наших полей. Уберем строчку `{{ form.as_p }}` и вместо нее запишем все поля формы по порядку, друг за другом.

Первое поле **title** мы сформируем так:

```
<label class="form-label" for="{{
form.title.id_for_label }}">{{form.title.label}}:
</label>{{ form.title }}</p>
<div class="form-error">{{ form.title.errors }}</div>
```

Мы здесь самостоятельно прописали HTML-теги внутри формы. Сначала идет тег абзаца `<p>`, внутри него тег `<label>` для оформления подписи. У нее указан класс оформления **form-label** и идентификатор через свойство **form.title.id\_for\_label**. Далее, идет само название **form.title.label** и после тега `<label>` отображается поле для ввода заголовка **form.title**. Вот так можно самостоятельно расписать атрибуты объекта **form** внутри шаблона. Следующая строка определяет тег `<div>` с классом оформления **form-error** для отображения возможных ошибок при вводе некорректных данных. Список ошибок доступен через переменную **form.title.errors**.

Если теперь обновить страницу сайта, то увидим это одно поле в форме. По аналогии можно прописать и все остальные поля:

```
<label class="form-label" for="{{
form.slug.id_for_label }}">{{form.slug.label}}:
</label>{{ form.slug }}</p>
<div class="form-error">{{ form.slug.errors }}</div>
<label class="form-label" for="{{
form.content.id_for_label }}">{{form.content.label}}:
</label>{{ form.content }}</p>
<div class="form-error">{{ form.content.errors }}</div>
<label class="form-label" for="{{
form.is_published.id_for_label
}}">{{form.is_published.label}}: </label>{{
form.is_published }}</p>
<div class="form-error">{{ form.is_published.errors
}}</div>
<label class="form-label" for="{{ form.cat.id_for_label
}}">{{form.cat.label}}: </label>{{ form.cat }}</p>
<div class="form-error">{{ form.cat.errors }}</div>
<label class="form-label" for="{{
form.husband.id_for_label }}">{{form.husband.label}}:
</label>{{ form.husband }}</p>
<div class="form-error">{{ form.husband.errors }}</div>
```

А в самом верху добавим строку:

```
<div class="form-error">{{ form.non_field_errors
}}</div>
```

для вывода ошибок валидации, не связанных с заполнением того или иного поля.

Это довольно гибкий вариант представления формы, и здесь можно очень тонко настроить отображение каждого поля. Однако объем кода при этом резко возрастает. В большинстве случаев все это можно существенно сократить. Все эти строки, в общем-то, повторяются, а, значит, их можно сформировать через цикл, следующим образом:

```
<div class="form-error">{{ form.non_field_errors
}}</div>
{% for f in form %}
<label class="form-label" for="{{ f.id_for_label
}}">{{f.label}}: </label>{{ f }}</p>
<div class="form-error">{{ f.errors }}</div>
{% endfor %}
```

Мы здесь на каждой итерации имеем объект поля формы и обращаемся к нужным его атрибутам. Такая запись значительно короче и гибче в плане изменения формы (достаточно поменять класс формы и это сразу изменит ее вид в шаблоне). Правда, все поля теперь будут иметь одни и те же стили оформления.

Однако, для виджетов стили оформления можно прописывать непосредственно в классе формы. Например, у класса поля ввода **title** добавить именованный параметр **widget**:

```
title = forms.CharField(max_length=255,  
label="Заголовок",  
widget=forms.TextInput(attrs={'class': 'form-input'}))
```

Мы здесь формируем виджет через класс **TextInput** и указываем у него стиль оформления **form-input**. При обновлении страницы видим, что первое поле изменило свой вид. И так можно делать со всеми полями. В частности, настроить размер поля **Textarea**:

```
content =  
forms.CharField(widget=forms.Textarea(attrs={'cols':  
50, 'rows': 5})), required=False, label="Контент")
```

То есть с помощью словаря **attrs** можно назначать любые атрибуты HTML для соответствующих тегов.

### Тестирование формы

Если мы попробуем отправить пустую форму на сервер, то браузер укажет, что поле **title** обязательное. То же самое и для поля URL, напомним что-нибудь латинскими буквами. Выберем категорию и нажмем «Добавить». В результате, в консоли у нас отображается словарь с принятыми данными:

```
{'title': 'ghfh', 'slug': 'gfhgfhfg', 'content': '', 'is_published': True, 'cat': <Category:  
Актрисы>, 'husband': None}
```

Все это мы можем сохранить в БД и сформировать новый пост. Если же данные в форме окажутся некорректными, например, в поле URL напишем что-то русскими буквами и сделаем отправку. Видим сообщение:

«Значение должно состоять только из латинских букв...»

и в консоли нет отображения данных, то есть, проверка не прошла. Вот так автоматически Django позволяет выполнять проверки на валидность заполненных данных.

Конечно, помимо используемых параметров в классах имеется множество других. Их можно посмотреть либо на странице документации:

<https://docs.djangoproject.com/en/4.2/ref/forms/fields/>

либо в самом PyCharm перейти в определение класса поля и прочитать о возможных используемых переменных в инициализаторе.

### Добавление новой записи

Теперь, когда наша форма готова, выполним добавление записи в БД. Для этого после проверки валидности данных, запишем конструкцию:

```
if form.is_valid():
    #print(form.cleaned_data)
    try:
        Women.objects.create(**form.cleaned_data)
        return redirect('home')
    except:
        form.add_error(None, 'Ошибка добавления поста')
```

Мы здесь используем ORM Django для формирования новой записи в таблице **women** и передаем в метод **create()** распакованный словарь полученных данных. Так как метод **create()** может генерировать исключения, то помещаем его вызов в блок **try** и при успешном выполнении, осуществляется перенаправление на главную страницу. Если же возникли какие-либо ошибки, то попадаем в блок **except** и формируем общую ошибку для ее отображения в форме.

Для начала введем корректные данные в форму, тогда после нажатия на кнопку «Добавить» в таблице **women** появится новая запись с заполненными полями. Вернемся в форму и попробуем добавить статью с неуникальным слагом. Тогда возникнет исключение, и мы увидим сообщение «Ошибка добавления поста».

Это был пример использования формы, несвязанной с моделью. В результате нам пришлось в классе **AddPostForm** дублировать поля, описанные в модели **Women** и, кроме того, вручную выполнять сохранение данных в таблицу **women**. Все это можно автоматизировать, используя форму в связке с моделью.

### Валидация полей формы

Мы создали форму для добавления новых статей на сайт. Каждое поле проверяется на корректность (валидность) данных, прежде чем они попадут в

БД. Отметим, что классы полей формы имеют больше встроенных параметров, которыми можно задавать ограничения, чем классы моделей. Например, помимо **max\_length** в классе **CharField** можно прописать и параметр **min\_length**, задающий минимальное значение. Например, так:

```
title = forms.CharField(max_length=255, min_length=5,
label="Заголовок",
widget=forms.TextInput(attrs={'class': 'form-input'}))
```

В результате сформируется ограничение для **min\_length** и на уровне HTML-тега и на уровне фреймворка Django, если переданная длина заголовка окажется меньше 5 символов.

Если нас не устраивает сообщение об ошибке по умолчанию для тех или иных ограничений, то мы можем сформировать свои, используя словарь **errors** отдельно для каждого поля. Например, так:

```
title = forms.CharField(max_length=255,
min_length=5, label="Заголовок",
widget=forms.TextInput(attrs={'class': 'form-input'}),
error_messages={
    'min_length': 'Слишком
короткий заголовок',
    'required': 'Без
заголовка - никак',
})
```

Соответственно, у других классов свои параметры для задания ограничений. Все их можно посмотреть или на странице документации:

<https://docs.djangoproject.com/en/4.2/ref/forms/fields/>

или перейти в сам класс поля, посмотреть в инициализаторе, а также в инициализаторе базового класса.

### Дополнительные валидаторы

Каждый класс поля формы и модели имеет атрибут **validators**, позволяющий указывать список классов-валидаторов для предобработки переданных данных на стороне сервера. Набор стандартных таких классов доступен по ветке **django.core.validators**. Например, импортируем два валидатора:

```
from django.core.validators import MinLengthValidator,
MaxLengthValidator
```

и пропишем их для поля `slug` следующим образом:

```
slug = forms.SlugField(max_length=255, label="URL",
validators=[
    MinLengthValidator(5),
    MaxLengthValidator(100),
])
```

Переходим на страницу добавления поста и введем слаг меньше 5 символов. При отправке данных увидим сообщение от валидатора `MinLengthValidator`.

Если стандартные сообщения нас не устраивают, то мы можем определить свои следующим образом:

```
slug = forms.SlugField(max_length=255, label="URL",
validators=[
    MinLengthValidator(5, message="Минимум 5
СИМВОЛОВ"),
    MaxLengthValidator(100, message="Максимум 100
СИМВОЛОВ"),
])
```

Конечно, похожего результата мы могли бы достичь, если бы прописали у класса **Slug** параметр `min_length=5`. Но на все случаи жизни параметры не определишь. Они охватывают лишь наиболее частые ситуации. Например, в классах модели нет параметра `min_length`, но мы можем указать валидатор **MinLengthValidator**. Например, в классе модели **Women** для поля **slug** указать такие же валидаторы:

```
slug = models.SlugField(max_length=255,
db_index=True, unique=True, validators=[
    MinLengthValidator(5),
    MaxLengthValidator(100),
])
```

Они будут использоваться при проверке перед записью данных в БД.

### Создание пользовательских валидаторов

По аналогии с существующими классами валидации мы можем объявить свои собственные. Если посмотреть на имеющиеся классы, то можно заметить, что они должны содержать магический метод `__call__()`, в котором выполняется проверка. Если она не проходит, то генерируется исключение **ValidationError** с выдаваемым сообщением.

Наш валидатор будет проверять, чтобы в данных присутствовали только русские символы, цифры, дефис и пробел. Другие символы будем считать недопустимыми. Для этого определим валидатор следующим образом:

```
@deconstructible
class RussianValidator:
    ALLOWED_CHARS =
    "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдеёжзийклмнопрсту
    фхцчшщбыъёя0123456789- "
    code = 'russian'

    def __init__(self, message=None):
        self.message = message if message else "Должны
        присутствовать только русские символы, дефис и пробел."

    def __call__(self, value):
        if not (set(value) <= set(self.ALLOWED_CHARS)):
            raise ValidationError(self.message,
            code=self.code, params={"value": value})
```

Мы здесь воспользовались декоратором **deconstructible**, так как он присутствует у всех классов-валидаторов. Затем мы можем при создании экземпляра этого класса указать сообщение об ошибке, либо будет использоваться стандартное. И, самое главное, это магический метод **\_\_call\_\_()**. Напомним, что он работает, когда объект класса **RussianValidator** вызывается подобно функциям с круглыми скобками. Соответственно, в него передается значение поля, в котором этот валидатор будет определен. Внутри этого метода идет проверка на допустимые символы и если находится хотя бы один недопустимый, то генерируется исключение **ValidationError** с сообщением об ошибке.

Давайте подключим этот валидатор к полю **title** класса **AddPostForm**:

```
class AddPostForm(forms.Form):
    title = forms.CharField(max_length=255,
    min_length=5, label="Заголовок",

                                widget=forms.TextInput(attrs={'cla
    ss': 'form-input'}),

                                validators=[
                                    RussianValidator(),
                                ],
                                error_messages={
                                    'min_length': 'Слишком
    короткий заголовок',
```



```

                                'required': 'Без
заголовка - никак',
                                })
...

```

Перейдем на страницу добавления поста и введем в заголовке латинские символы. При отправке формы сработает наш валидатор и выдаст сообщение об ошибке.

Создание своих классов валидаторов имеет смысл только в том случае, если предполагается его многократно использовать в текущем проекте. Если же нам требуется выполнить такую проверку только для одного поля **title**, то делается это гораздо проще. В самом классе **AddPostForm** можно объявить метод, который начинается с ключевого слова **clean\_** и далее имя поля для валидации. Например, для поля **title**, следует объявить метод с именем **clean\_title** следующим образом:

```

def clean_title(self):
    title = self.cleaned_data['title']
    ALLOWED_CHARS =
"АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдеёжзийклмнопрсту
фхцчшщбыьёя0123456789- "
    if not (set(title) <= set(ALLOWED_CHARS)):
        raise ValidationError("Должны быть только
русские символы, дефис и пробел.")

    return title

```

Получаем в итоге тот же самый эффект.

Механизм валидации данных работает следующим образом. Сначала данные формы отправляются на сервер, там проверяются сначала стандартными валидаторами. Если эта проверка прошла, то только после этого вызываются пользовательские валидаторы, прописанные в классе формы или подключенные через параметр **validator**.

### Формы, связанные с моделями

Когда форма предполагает тесное взаимодействие с какой-либо моделью, то лучше ее напрямую с ней и связать.

Перейдем в файл **women/forms.py** и класс **AddPostForm** унаследуем от другого базового класса **ModelForm**. А внутри дочернего класса объявим вложенный класс **Meta** с атрибутами **model** и **fields**:

```

class AddPostForm(forms.ModelForm):

```

```
class Meta:
    model = Women
    fields = '__all__'
```

Атрибут **model** устанавливает связь формы с моделью **Women**, а свойство **fields** – определяет поля для отображения в форме. Значение **\_\_all\_\_** указывает показывать все поля, кроме тех, что заполняются автоматически. В результате, мы увидим уже готовую форму, только без полей **time\_create** и **time\_update**, так как они наполняются без участия пользователя.

Однако, на практике рекомендуется явно прописывать список полей, необходимых для отображения в форме. В нашем случае это будет выглядеть так:

```
fields = ['title', 'slug', 'content', 'is_published',
'cat', 'husband', 'tags']
```

Затем, чтобы описать стили оформления для каждого поля, используется атрибут **widgets** класса **Meta**:

```
class Meta:
    model = Women
    fields = ['title', 'slug', 'content',
'is_published', 'cat', 'husband', 'tags']
    widgets = {
        'title': forms.TextInput(attrs={'class':
'form-input'}),
        'content': forms.Textarea(attrs={'cols':
60, 'rows': 10}),
    }
```

Обновляем страницу и видим, что эти свойства были применены к указанным полям.

Установим также у списков свойства **empty\_label**. Для этого явно пропишем атрибуты для полей выбора, как это мы делали в предыдущей модели:

```
class AddPostForm(forms.ModelForm):
    cat =
forms.ModelChoiceField(queryset=Category.objects.all(),
empty_label="Категория не выбрана", label="Категории")
    husband =
forms.ModelChoiceField(queryset=Husbands.objects.all(),
required=False, empty_label="Не замужем", label="Муж")
```

```

class Meta:
    model = Women
    fields = ['title', 'slug', 'content',
'is_published', 'cat', 'husband', 'tags']
    labels = {'slug': 'URL'}
    widgets = {
        'title': forms.TextInput(attrs={'class':
'form-input'}),
        'content': forms.Textarea(attrs={'cols':
50, 'rows': 5}),
    }

```

Теперь наша форма ничем не отличается от предыдущего варианта. Более того, в ней появился метод **save()**, который сохраняет переданные данные в БД. Воспользуемся им, перейдем в файл **women/views.py** и в функции представления **addpage()** вместо прежней конструкции:

```
Women.objects.create(**form.cleaned_data)
```

запишем:

```
form.save()
```

Форма готова к использованию. Фреймворк Django позволяет предельно автоматизировать весь процесс создания и обработки данных форм. Давайте убедимся, что все работает. Перейдем на страницу:

<http://127.0.0.1:8000/addpage/>

Здесь специально указан неуникальный URL, и для формы, связанной с моделью, мы получаем встроенное сообщение о проблеме. То есть метод **save()** берет на себя всю проверку корректности записи данных и блок **try-except** нам уже не нужен. Уберем его. Введем уникальный URL, и пост успешно добавляется в БД.

Этот пример показывает, насколько упрощается взаимодействие между пользователем и БД, с использованием форм, связанных с моделью.

### Создание собственных валидаторов формы

Если стандартных проверок для валидации полей формы недостаточно, то мы, по-прежнему, можем формировать свои собственные валидаторы, так же, как и у форм, не связанных с моделью. Например, опишем валидатор для поля **title**, который бы не позволял вводить строку более 50 символов. В SQLite эта проверка не проходит, а просто обрезается заголовок до указанной

длины, а мы сделаем так, что пользователю будет показываться сообщение, что название статьи слишком большое.

Для этого в форме **AddPostForm** объявим метод с именем **clean\_title** и следующим содержимым:

```
def clean_title(self):
    title = self.cleaned_data['title']
    if len(title) > 50:
        raise ValidationError('Длина превышает 50
СИМВОЛОВ')

    return title
```

Мы здесь сначала считываем заголовок, переданный из формы, из словаря очищенных данных – **cleaned\_data**. Затем проверяем, если длина больше 50 символов, то пользователю будет показываться сообщение «Длина превышает 50 символов». Иначе, возвращается заголовок **title**, который и будет помещен в БД.

Проверим, как это работает. Введем заголовок длиной более 50 символов, заполним другие поля и при нажатии на кнопку «Добавить» увидим искомое сообщение. Мы сделали свой собственный валидатор для поля **title** на уровне формы. По аналогии, можно создавать другие валидаторы для остальных полей, если в этом возникает необходимость.

### Загрузка (upload) файлов на сервер

Посмотрим, как можно на сервер загружать отдельные файлы. Создадим HTML-форму для загрузки произвольных файлов. Прикрепим ее к уже существующему маршруту:

<http://127.0.0.1:8000/about/>

с отображением шаблона **about.html**. Откроем этот шаблон и после заголовка **h1** пропишем тег **form** следующим образом:

```
<form action="" method="post" enctype="multipart/form-
data">
    {% csrf_token %}
    <p ><input type="file" name="file_upload"></p>
    <p ><button type="submit">Отправить</button></p>
</form>
```

Обратите внимание, что в форме обязательно должен быть прописан параметр **enctype="multipart/form-data"**. Без него файлы загружаться на

сервер не будут. Также должно быть определено специальное поле **input** с типом **file** и стандартная кнопка формы для отправки данных. Это минимальный набор элементов для выбора и отправки произвольного файла на сервер.

В качестве обработчика файла у нас сейчас определена функция представления **about()**. Что будет ей передаваться? При отправке файла в словаре **FILES** появляется ключ **file\_upload**, который ссылается на объект загруженных данных файла. Нам осталось обратиться к этому объекту и сохранить данные в файл на сервер. Согласно документации:

<https://docs.djangoproject.com/en/4.2/topics/http/file-uploads/>

у этого объекта есть методы **read()** и **chunks()**. Первый читает все данные целиком, а второй по частям. Конечно, предпочтительно использовать второй метод, так как файлы могут быть больших объемов. Ниже приведена функция, которая сохраняет загруженные данные в файл с помощью метода **chunks()**. Скопируем ее и вставим перед функцией представления **about()**:

```
def handle_uploaded_file(f):
    with open(f"uploads/{f.name}", "wb+") as
destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

И изменим путь загрузки, а затем вызовем ее при отправке POST-запроса:

```
def about(request):
    if request.method == "POST":
        handle_uploaded_file(request.FILES['file_upload'])

    return render(request, 'women/about.html',
{'title': 'О сайте', 'menu': menu})
```

Попробуем выполнить загрузку. Выберем файл, нажимаем кнопку «Отправить» и видим ошибку, что каталог **uploads**, куда мы собираемся поместить файл, не существует. Добавим его в корень нашего проекта. Теперь при отправке файла никаких ошибок не возникает, и мы увидим файл в директории **uploads**.

### Класс поля FileField

Мы сделали простейший вариант загрузки произвольных файлов на сервер. Конечно, у него есть недостатки. Например, при отправке формы без выбранного файла возникает ошибка. То есть поля формы мы сейчас никак

не проверяем на корректность (валидность). Но, как мы уже знаем, это автоматизируется фреймворком Django при использовании классов форм. Давайте воспользуемся формой, не привязанной к модели.

Перейдем в файл **women/forms.py** и в нем объявим еще один класс:

```
class UploadFileForm(forms.Form):  
    file = forms.FileField(label="Файл")
```

Затем создадим форму в функции представления **about()** следующим образом:

```
def about(request):  
    if request.method == "POST":  
        form = UploadFileForm(request.POST,  
request.FILES)  
        #  
        handle_uploaded_file(request.FILES['file_upload'])  
    else:  
        form = UploadFileForm()  
  
    return render(request, 'women/about.html',  
{ 'title': 'О сайте', 'menu': menu, 'form': form })
```

И отобразим ее в шаблоне **about.html**:

```
<form action="" method="post" enctype="multipart/form-  
data">  
    {% csrf_token %}  
    {{ form.as_p }}  
    <button type="submit">Отправить</button></p>  
</form>
```

Переходим по адресу:

<http://127.0.0.1:8000/about/>

и видим форму, похожую на прежнюю. Здесь можно выбрать файл, отправить, но сохраняться на сервере он пока не будет. Мы этот функционал еще не прописали. Добавим в функцию **about()** проверку полей и вызов функции сохранения файла, получим:

```
def about(request):  
    if request.method == "POST":  
        form = UploadFileForm(request.POST,  
request.FILES)
```

```

        if form.is_valid():

handle_uploaded_file(form.cleaned_data['file'])
        else:
            form = UploadFileForm()

        return render(request, 'women/about.html',
{'title': 'О сайте', 'menu': menu, 'form': form})

```

Обратите внимание, что при вызове метода **is\_valid()** выполняется проверка полей формы, и, если попытаться отправить форму без файла, то Django обработает эту ситуацию и возвратит сообщение об ошибке обратно в форму. Это очень удобно. Все, что нам остается, это получить объект-файл и сохранить его в каталог **uploads**. Причем здесь мы обращаемся уже к словарию **cleaned\_data** с набором проверенных и корректных данных.

Сейчас имеется еще один серьезный недостаток при загрузке файлов. Если загружать разные файлы, но с одинаковыми именами, то они будут перезаписывать друг друга в папке **uploads**. Давайте поправим этот момент. Для этого воспользуемся специальным модулем **uuid** (в файле **women/views.py**):

```
import uuid
```

И в функции **handle\_uploaded\_file()** будем генерировать случайные имена для загружаемых файлов:

```

def handle_uploaded_file(f):
    name = f.name
    ext = ''

    if '.' in name:
        ext = name[name.rindex('.'):]
        name = name[:name.rindex('.')]

    suffix = str(uuid.uuid4())
    with open(f"uploads/{name}_{suffix}{ext}", "wb+")
as destination:
        for chunk in f.chunks():
            destination.write(chunk)

```

Теперь вероятность появления файлов с одинаковыми именами практически равна нулю. Для надежности дополнительно можно файлы загружать в подкаталоги с годом, месяцем и днем.

Очень часто на практике требуется загружать не произвольные файлы, а только графические. Для этого в форме предусмотрено специальное поле



**ImageField**. Оно является расширением класса **FileField** и позволяет выбирать исключительно графические файлы. Чтобы им воспользоваться нам достаточно его прописать в классе формы:

```
class UploadFileForm(forms.Form):  
    file = forms.ImageField(label="Изображение")
```

Обработчик остается неизменным.

### Загрузка файлов с использованием моделей

Мы научились загружать произвольные и графические файлы на сервер. Но непонятно, как воспользоваться этими файлами, если отсутствует информация: кто их загружал, для чего загружал, какое имя файла в итоге было сгенерировано и т.п. Поэтому, чаще всего, загрузка происходит в связке с моделями, то есть с таблицами БД. Давайте разберемся, как это работает.

Для простоты сначала определим модель новой таблицы, в которой будут храниться ссылки на загруженные файлы. Пусть она будет следующей (в файле **women/models.py**):

```
class UploadFiles(models.Model):  
    file = models.FileField(upload_to='uploads_model')
```

Здесь фигурирует уже знакомый нам класс **FileField**, но взятый уже из ветки **models**. Это несколько иной класс, и у него имеется дополнительный параметр **upload\_to** – каталог, в который будет происходить загрузка файлов.

После определения новой модели нам нужно создать и применить миграции:

```
python manage.py makemigrations
```

```
python manage.py migrate
```

В БД появляется новая таблица, и мы напрямую воспользуемся нашей моделью для сохранения файла на сервер. Для этого перейдем в файл **women/views.py** и найдем функцию **about()**. В ней после проверки корректности переданных данных создадим объект класса модели **UploadFiles** с последующим вызовом метода **save()**:

```
def about(request):  
    if request.method == "POST":  
        form = UploadFileForm(request.POST,  
request.FILES)  
        if form.is_valid():
```

```
#
handle_uploaded_file(form.cleaned_data['file'])
    fp =
UploadFiles(file=form.cleaned_data['file'])
    fp.save()
else:
    form = UploadFileForm()

return render(request, 'women/about.html',
{'title': 'О сайте', 'menu': menu, 'form': form})
```

Посмотрим, как это будет работать. Выберем файл, нажмем «Отправить», и у нас автоматически создается каталог **uploads\_model** и в нем размещен загруженный файл. А в таблице БД появилась запись с путем к этому файлу. Если мы попробуем еще раз загрузить тот же файл, то никаких коллизий не возникнет, и ему автоматически будет присвоено другое имя. Соответственно, в таблице увидим еще одну запись.

Вот так фреймворк Django позволяет автоматизировать процесс загрузки файлов на сервер. Функция **handle\_uploaded\_file()** нам теперь не нужна, и ее можно удалить.

Осталось сделать один важный штрих. Мы можем на уровне всего проекта указать единую папку, в которую будут сохраняться все загружаемые файлы. Это делается в файле **settings.py** пакета конфигурации, в котором можно определить следующую переменную:

```
MEDIA_ROOT = BASE_DIR / 'media'
```

Теперь все файлы будут загружаться в каталог **media**, создавая в нем подкаталоги. Проверим это. Удалим все папки **uploads**, перезапустим веб-сервер и снова загрузим файл. У нас автоматически сформируется каталог **media**, в нем подкаталог **uploads\_model**, указанный в параметре **upload\_to='uploads\_model'** класса модели и в этом подкаталоге размещен выбранный файл. В таблице увидим очередную запись с этой последней загрузкой.

### Добавление изображений к постам

Пока мы использовали обычную форму для загрузки файлов. Однако, совместно с моделями было бы логично применять формы в связке с ними, то есть класс **ModelForm**. Давайте это сделаем.

У нас уже есть форма добавления поста по известным женщинам, которая базируется на модели **Women**. Поэтому просто добавим в эту модель еще одно поле **photo** следующим образом:

```
photo = models.ImageField(upload_to="photos/%Y/%m/%d/",
default=None, blank=True, null=True,
verbose_name="Фото")
```

Мы здесь используем уже знакомый нам класс **ImageField** с параметром **upload\_to**, который определяет путь загрузки для изображений. Обратите внимание, как он определен. Здесь на место спецификаторов %Y, %m, %d будут подставлены текущий год (четыре цифры), текущий месяц и текущий день. Это позволит нам при массовой загрузке изображений, разнести их по отдельным подкаталогам.

После изменения модели создадим и выполним миграции:

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

Затем перейдем в шаблон **addpage.html**. Здесь в тег формы необходимо добавить параметр:

```
enctype="multipart/form-data"
```

Иначе связанные данные не будут отправляться браузером на сервер.

В файле **women/forms.py** в классе **AddPostForm** укажем поле **photo** в коллекции **fields**:

```
fields = ['title', 'slug', 'content', 'photo',
'is_published', 'cat', 'husband', 'tags']
```

А в файле **women/views.py** в функции **addpage()** при создании модели **AddPostForm** по ветке **POST** укажем второй параметр **request.FILES**:

```
def addpage(request):
    if request.method == 'POST':
        form = AddPostForm(request.POST, request.FILES)
        if form.is_valid():
            # print(form.cleaned_data)
            form.save()
            return redirect('home')
    else:
        form = AddPostForm()

    return render(request, 'women/addpage.html',
{'menu': menu, 'title': 'Добавление статьи', 'form':
form})
```

Теперь все готово, запускаем веб-сервер, переходим на страницу добавления поста:

<http://127.0.0.1:8000/addpage/>

и в форме видим отображение поля для загрузки файла изображения. Давайте создадим какой-нибудь новый пост вместе с изображением:

Рианна; rianna; Биография Рианны; Певицы

На сервере в рабочем каталоге появится папка **media** с набором подкаталогов и загруженным файлом изображения. В таблице **women** видим новую запись с заполненным полем **photo**.

Мы организовали загрузку изображения для поста и связали его с конкретной записью с помощью дополнительного поля **photo**.

### Отображение загруженных изображений в HTML-документе

Теперь настроим показ изображения при просмотре поста. Но сначала посмотрим, что собой представляет поле **photo**.

Перейдем в консоль фреймворка Django:

```
python manage.py shell_plus
```

И прочитаем из таблицы **women** последнюю запись:

```
w = Women.objects.all()[0]
```

У объекта **w** имеется атрибут **photo**:

```
w.photo
```

который ссылается на объект **ImageFieldFile**. Если после **photo** поставить точку и нажать **Tab**, то увидим множество атрибутов и методов этого объекта. В частности, нас будет интересовать локальный атрибут **url**:

```
w.photo.url
```

который возвращает URL-адрес для получения изображения с сервера. Воспользуемся им и пропишем в шаблоне **post.html** следующие строки сразу после заголовка первого уровня:

```
{% if post.photo %}
</p>
```

```
{% endif %}
```

Однако, если сейчас открыть последний пост, то увидим, что изображение для него не загружается. Дело в том, что пока сервер не может связать этот URL с физическим маршрутом к файлу изображения на сервере. Для этого нам нужно сделать следующие действия. В файле конфигурации **settings.py** пропишем еще один параметр:

```
MEDIA_URL = '/media/'
```

Если сейчас обновить страницу поста, то URL-маршрут к изображению будет иметь этот префикс **media**. Зачем он нужен? Благодаря единому уникальному префиксу для всех подгружаемых медиа-файлов, мы можем в **urls.py** пакета конфигурации определить для них маршрут выдачи этих файлов. Для этого пропишем такие строки:

```
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

То есть, если веб-сервер работает в режиме отладки, то в список маршрутов нужно добавить еще один с префиксом **media** и связать его с каталогом **MEDIA\_ROOT**, где эти файлы расположены. Только так в режиме отладки мы можем раздавать загруженные файлы. В боевом режиме в этом нет необходимости, т.к. сервер уже будет иметь необходимые настройки для связки маршрута с рабочим каталогом.

Обновляем страницу и видим, что теперь все работает корректно.

По аналогии сделаем показ изображений в списке постов. Для этого перейдем в шаблон **index.html** и перед заголовком **h2** пропишем следующие строки:

```
{% if p.photo %}
    <p ></p>
{% endif %}
```

Переходим на главную страницу, видим изображение в списке постов.

### Отображение изображений в админ-панели

Следующим шагом добавим отображение миниатюр изображений, связанных с нашими постами, непосредственно в списке. Перейдем в файл **women/admin.py** и в классе **WomenAdmin** в списках **fields** и **list\_display** укажем поле **photo**:

```
fields = ['title', 'slug', 'content', 'photo', 'cat',
          'husband', 'tags']
list_display = ('title', 'photo', 'time_create',
               'is_published', 'cat')
```

А поле **brief\_info** удалим.

Переходим в админ-панель:

<http://127.0.0.1:8000/admin/>

и в списке женщин в поле **photo** отображается путь к изображению. Но мы бы хотели здесь видеть именно изображение, а не путь к нему. Для этого сформируем новое вычисляемое поле, по аналогии с полем **brief\_info**. Переименуем метод **brief\_info** в **post\_photo** и возвратим строку в виде тега **img**:

```
@admin.display(description="Изображение")
def post_photo(self, women: Women):
    return mark_safe(f"<img src='{women.photo.url}'
width=50>")
```

Здесь функция **mark\_safe()** используется для того, чтобы теги не экранировались и отработывали как HTML-теги. Осталось в список **list\_display** вместо **photo** указать **post\_photo**:

```
list_display = ('title', 'post_photo', 'time_create',
               'is_published', 'cat')
```

Если сейчас обновить список в админ-панели, то увидим ошибку. Она связана с тем, что не у всех постов имеются изображения. Поэтому нужно прописать следующую проверку:

```
@admin.display(description="Изображение")
def post_photo(self, women: Women):
    if women.photo:
        return mark_safe(f"<img
src='{women.photo.url}' width=50>")
    return "Без фото"
```

Теперь никаких ошибок нет, все работает, как требуется.

Более того, мы прямо здесь, в админ-панели, можем теперь редактировать посты и назначать им изображения. Также можно сделать отображение миниатюры непосредственно при изменении статьи. Для этого достаточно указать ссылку на наш метод **post\_photo()** в атрибутах:

```
@admin.register(Women)
class WomenAdmin(admin.ModelAdmin):
    fields = ['title', 'slug', 'content', 'photo',
              'post_photo', 'cat', 'husband', 'tags']
    readonly_fields = ['post_photo']
    ...
```

Теперь мы видим изображение загруженной фотографии в форме редактирования.

Чтобы узнать о других возможностях настройки формы редактирования, на странице документации можно посмотреть список атрибутов для класса **ModelAdmin**. В частности, установка вот такого атрибута:

```
save_on_top = True
```

добавляет верхнюю панель для управления записью, что бывает очень удобно.

### Задание

1. Создать форму для добавления записи, несвязанную с моделью. Проверить работу формы на корректных и некорректных данных с использованием стандартных валидаторов. Написать собственный валидатор, проверить работу формы на нем.
2. Создать форму для добавления записи, связанную с моделью. Проверить работу формы на корректных и некорректных данных с использованием стандартных валидаторов. Написать собственный валидатор, проверить работу формы на нем.
3. Реализовать загрузку файлов на сервер. Использовать генерацию случайных имен для обеспечения возможности загрузки файлов с одинаковыми исходными именами.
4. Добавить изображения к записям в БД. Добавить соответствующее поле в форму добавления записи. Проверить работу формы. Обеспечить отображение изображений на сайте и в админ-панели.