Формы, связанные с моделями. Пользовательские валидаторы

Смотреть материал на видео

Архив проекта: lesson-14-coolsite.zip

На предыдущем занятии мы с вами познакомились с созданием форм, не связанных с моделями на примере добавления статьи. Это был несколько искусственный пример, так как добавление нового поста связано с обращением к БД. Из-за этого у нас, фактически, получилось дублирование кода: в классе формы AddPostForm мы прописывали аналогичные атрибуты, что и в классе модели Women. Это не очень хорошо и когда форма предполагает тесное взаимодействие с какой-либо моделью, то лучше ее напрямую с ней и связать. На этом занятии вы увидите, как это делается.

Перейдем в файл women/forms.py и класс AddPostForm унаследует 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 как раз устанавливает связь формы с моделью Women, а свойство fields – определяет поля для отображения в форме. Значение __all__ говорит 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 указывать список полей, необходимых для отображения в форме. В нашем случае это будет следующий список:

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

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

```
    class Meta:
        model = Women
        fields = ['title', 'slug', 'content', 'is_published', 'cat']
        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.

По идее нам бы еще хотелось у списка установить свойство:

empty_label = "Категория не выбрана"

Для этого запишем конструктор у класса формы, и после вызова конструктора базового класса, присвоим атрибуту empty_label нужное значение:

```
class AddPostForm(forms.ModelForm):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fields['cat'].empty_label = "Категория не выбрана"
...
```

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

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

запишем:

```
form.save()
```

Все, форма готова к использованию! Видите, как 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/

Я специально укажу неуникальный URL и, смотрите, для формы связанной с моделью мы получаем четкое 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.

Загрузка изображений через форму

Во всех наших примерах этого и предыдущего занятия игнорировалось поле photo. Давайте отобразим и его, чтобы пользователь указывал изображение, связанное со статьей. Проще всего это сделать именно с формами, связанными с моделями. Сначала в атрибуте fields вложенного класса Meta формы AddPostForm, добавим в список поле photo:

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

Затем, в функции представления addpage, при создании экземпляра формы, вторым аргументом передадим список файлов, которые были переданы на сервер из формы:

```
form = AddPostForm(request.POST, request.FILES)
```

И последнее, что нам нужно сделать, в шаблоне addpage.html добавить в тег <form> атрибут:

enctype="multipart/form-data"

обязательный, в случае передачи каких-либо файлов совместно с данными полей. Все, обновляем страницу:

http://127.0.0.1:8000/addpage/

Заполняем поля, указываем картинку и все дальнейшие операции Django берет на себя: выполняет загрузку, передачу и проверку данных. Если все в порядке, то запись добавляется в БД.

Создание собственных валидаторов формы

Валидация полей формы AddPostForm выполняется согласно атрибутам, указанным в модели Women. Правда, используя СУБД SQLite, проверка на максимальную длину (атрибут max_length) не выполняется, строка просто обрезается. Это особенность конкретной СУБД. Другие типы БД будут выдавать сообщения, если длина превышена. Поэтому сейчас на этот момент просто не обращайте внимание. Другие указанные валидаторы в атрибутах модели, например, unique=True или blank=True, отрабатывают корректно.

Но, что если стандартных проверок недостаточно и возникает необходимость создать свой валидатор на уровне формы? Никаких проблем! Фреймворк Django позволяет нам это сделать. Давайте, как раз создадим валидатор для поля title, который бы не позволял вводит строку более 200 символов. Как я уже отметил 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.

Механизм работы собственных валидаторов следующий. Сначала данные формы, при отправке на сервер, проверяются стандартными валидаторами. Если эта проверка прошла, то вызываются пользовательские валидаторы, прописанные в классе формы. Что они из себя представляют? Это обычные методы, имена которых начинаются с префикса clean_ и, затем, прописывается имя поля для проверки. Например, мы хотим описать валидатор для поля title, значит, в форме AddPostForm нужно объявить метод с именем clean_title:

```
class AddPostForm(forms.ModelForm):
...
    def clean_title(self):
        pass
```

Этот метод должен генерировать исключение (обычно ValidationError) если поле title содержит недопустимые данные и возвращать заголовок, в противном случае. В нашем случае реализация метода clean_title будет такой:

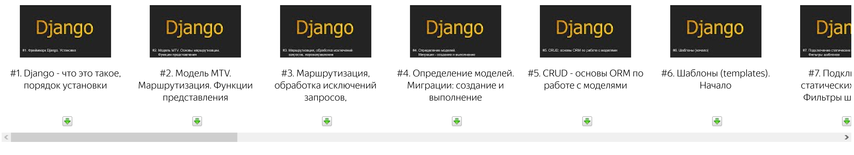
```
    def clean_title(self):
        title = self.cleaned_data['title']
        if len(title) > 200:
            raise ValidationError('Длина превышает 200 символов')

        return title
```

Мы здесь вначале считываем заголовок, переданный из формы, из словаря очищенных данных cleaned_data. Затем, проверяем, если длина больше 200 символов, то пользователю будет показываться сообщение «Длина превышает 200 символов». Иначе, возвращается заголовок title, который и будет помещен в БД.

Опять же, все предельно просто. Проверим, как это работает. Введем заголовок длиной более 200 символов, заполним другие поля и при нажатии на кнопку «Добавить» увидим искомое сообщение. Все, мы сделали свой собственный валидатор для поля title на уровне формы. По аналогии можно создавать другие валидаторы для остальных полей, если в этом возникнет необходимость.

Видео по теме



#1. Django - что это такое, порядок установки

#2. Модель MTV. Маршрутизация. Функции представления

#3. Маршрутизация, обработка исключений запросов.

#4. Определение моделей. Миграции: создание и выполнение

#5. CRUD - основы ORM по работе с моделями

#6. Шаблоны (templates). Начало

#7. Подкл статически Фильтры ш

← Предыдущая