Theory: Forms and validation

39 minutes 0 / 5 problems solved

Forms are the key elements of the user interface: they define what kind of behavior the service expects from the input fields. Knowing how to make easy and usable forms helps to keep the service simple and clear.

A form is a part of an HTML page, but controlling and validating the data is up to you. That's why Django provides tools to do it with Python. In this topic, we will inspect classes from the Django framework, types of input fields and the process of validation.

§1. Forms

This situation is familiar to many of us: there's a close friend who lives far away, someone from childhood or a pen-friend you've never actually met. Holidays come, and you want to surprise them with a postcard. How can you do it? Well, it so happens that we're working hard on the ***hypergift*** service. With this service, you can send a postcard to any place worldwide.

We start with this form:

```
from django import forms


class PostcardForm(forms.Form):
    address = forms.CharField(label='Destination Address')
    author = forms.CharField(min_length=3)
    compliment = forms.CharField(max_length=1024)
    date_of_delivery = forms.DateField(input_formats=['%Y/%m/%d'])
    email = forms.EmailField()
```

To make a custom form, we inherit the class from `django.forms.Form`. The class provides means to render the form in templates, validate it and show exact errors in the input. Fields should be declared with specific types for correct validation of the user's input (don't worry, we'll discuss it in more detail further on).

§2. Forms in Templates

A form is an element of an HTML page, and Django provides tools to render its fields correctly. It also gives default methods to format fields as a table, an unordered list or paragraphs.

Assume that we initialize a form with no arguments and save it in the `postcard_form` variable. We pass the context dictionary `{'postcard_form': postcard_form}` to a template and add csrf_token to prevent security issues:

```
<form method="post">{% csrf_token %}
  <table>{{ postcard_form.as_table }}</table>
  <button type="submit">Submit</button>
</form>
```

The method `postcard_form.as_table` converts the instance of the `PostcardForm` class to `<tr>`, `<th>` and `<td>` HTML tags with appropriate labels and attributes. The ***address*** field has a custom label in the class, so it also has the same value of the label on a page.

```
<th><label for="id_address">Destination Address:</label></th>
```

In the browser the form would look like this:

Current topic:

Forms and validation

Topic depends on:

- ✕ Django template language — Stage 2
- ✕ Submitting data — Stage 4

Topic is required for:

Registration and authentication

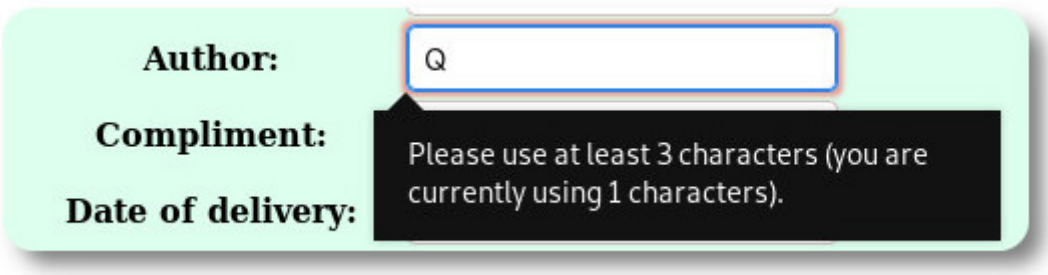
Table of contents:

Other methods for rendering forms are **form.as_p** (converts forms to paragraphs) and **form.as_ul (**converts them to an unordered list).

§3. Validation on the Client Side

To restrict the user's input, we use types in the form's fields. A form should be strict to catch typos and help the user put in data correctly. Also, the limitations will control for other little accidents: for example, it's highly unlikely that the user has a 10000-symbol long address name, so probably some keyboard keys got stuck.

The field ***author*** has a minimum length of three symbols by definition: no less than one character for the first name and two for the last name. What happens if a user tries to send two or fewer symbols?



The user sees a warning that the field should contain at least three characters. This check is done by the browser on a local computer as we haven't sent any data to the server yet. Let's look at the HTML syntax of an element more closely:

```
<td>
<input type="text" name="author" minlength="3" required id="id_author"></td>
```

The `<input>` element has a `minlength` attribute with the value "3" and an attribute `required`. It means that this field is mandatory and its length should be no less than 3 characters.

Another attribute related to the input length is **maxlength**. It sets the upper limit for the input length.

One more field worth double-checking is the ***email***: if it does not contain "@" and ".", it's definitely not right. As for more complex checks, that should happen on the server side.

§4. Validation on the Server Side

The browser is not almighty and cannot save the developers from bad input all the time. When an HTML element lacks attributes that specify the behavior, it's Django's turn to save the day.

There are no hints on a page as to which date format should be used. Any non-empty string will be valid, and you can see that in the HTML form:

```
<td>
<input type="text" name="date_of_delivery" required id="id_date_of_delivery"></td>
```

However, the field `date_of_delivery` has predefined `input_formats` that Django expects from an input. We can access sent data with the `POST` attribute of `request` in our handler and initialize an instance of `PostcardForm` with it:

```
postcard_form = PostcardForm(request.POST)
if postcard_form.is_valid():
    data = postcard_form.cleaned_data  # data is a regular dictionary
    ...
```

Magic happens when you call the `is_valid` method: all fields will be validated. After that, all valid fields will be accessed as a dictionary with `cleaned_data` property. If an input is correct, you can keep working with it. But how would the user know about a mistake in the input? The answer is: Django can signal about each one.

§5. Errors Rendering

We've already seen how useful messages are when it comes to properly filling the data. Now let's see how Django shows validation errors to the users. To make it work, pass the form initialized with `request.POST` to the context dictionary.

Assume that the user provides "2025-01-01" as a desired date of delivery:

Although they send the data, the server responds that the format is invalid. As you see, the default message is not very eloquent, so it's better to define a custom message:

```
postcard_form = PostcardForm(request.POST)
if postcard_form.is_valid():
    ...
elif 'date_of_delivery' not in postcard_form.cleaned_data:
    postcard_form.add_error('date_of_delivery', 'Use YYYY/MM/DD format')
```

If we don't have `date_of_delivery` in the validated data, we add a custom error message that suggests a hint:

Now the user knows what went wrong. Looks like we managed to ease the communication with the service. There's plenty of work left to do, but once we have learned how to use forms, the sky's the limit.

Report a typo

37 users liked this theory. **6** didn't like it. **What about you?**

Start practicing

Comments (8) Hints (2) Useful links (1) Show discussion