Kay forms manual

Overview

Kay has a utility for form handling. It is placed on kay.utils.forms, and kay.utils.forms.modelform. Here are conceptual elements for understanding Kay's form utilities.

- Widget

  A class that corresponds to HTML representation of a field, or even a form, e.g. <input type="text">, <textarea> or <form>...</form>. This class handles rendering HTML.

- Field

  A class that is responsible for doing validation, e.g. an FloatField that makes sure its data is a valid float value.

- Form

  A collection of fields that knows how to validate itself and convert itself to a widget.

Your First Form

Let's consider a form to implement "contact me" functionality.

```
from kay.utils import forms

class ContactForm(forms.Form):
  subject = forms.TextField(required=True, max_length=100)
  message = forms.TextField(required=True)
  sender = forms.EmailField(required=True)
  cc_myself = forms.BooleanField(required=False)
```

A form is composed of Field objects. In this case, our form has four fields: subject, message, sender and cc_myself. TextField, EmailField and BooleanField are just three of the available field types; a full list can be found in the document titled 'Form Field'.

If your form is going to be used to directly add or edit an AppEngine Datastore model, you can use a ModelForm to avoid duplicating your model description.

Using a form in a view

The standard pattern for processing a form in a view looks like this:

```
def contact(request):
  form = ContactForm()
  if request.method == "POST":
    if form.validate(request.form):
      # process the data
      # ...
      return redirect("/thanks/")
  return render_to_response("myapp/contact.html", {"form": form.as_widget()})
```

There are three code paths here:

1. If the form has not been submitted, a form instance of ContactForm is created and the widget instance is passed to the template.
2. If the form has been submitted, the data is validated using form.validate(request.form). If the submitted data is valid, it is processed and the user is re-directed to a "/thanks/" page.

3. If the submitted data is invalid, the widget instance created using form.as_widget() is passed to the template.

Processing the data form a form

Once form.validate() returns True, you can process the form submission safely in the knowledge that it confirms to the validation rules defined by your form. While you could access request.form directly at this point, it is better to access form.data or access the data in following style: form["subject"], form["message"] or form["sender"]. This data has not only been validated but will also be converted into the relevant Python types for you. In the above example, cc_myself will be a boolean value. Likewise, fields such as IntegerField and FloatField convert values to a Python int and float respectively.

Extending the above example, here's how the form data could be processed:

```
if form.validate(request.form):
  recipients = ["info@example.com"]
  if form["cc_myself"]:
    recipients.append(form["sender"])
  from google.appengine.api import mail
  mail.send_mail(sender=form["sender"], to=recipients,
                 subject=form["subject"], body=form["message"])
  return redirect("/thanks/")
```

Displaying a form using a template

Form widgets are very easy to render. In the above example, we passed our ContactForms's widget representation to the template using the context variable form. Here's a simple example template:

```
<body>
  {{ form()|safe }}
</body>
```

Widgets are callable, and if you call it, you can get the rendered HTML form. The result is already HTML escaped, so you need to append a safe filter after it. Here's the output for our example template:

```
<form action="" method="post">
  <div style="display: none">
    <input type="hidden" name="_csrf_token" value="c345asdf.........">
  </div>
  <dl>
    <dt><label for="f_subject">Subject</label></dt>
    <dd><input type="text" id="f_subject" value="" name="subject"></dd>
    <dt><label for="f_message">Message</label></dt>
    <dd><input type="text" id="f_message" value="" name="message"></dd>
    <dt><label for="f_sender">Sender</label></dt>
    <dd><input type="text" id="f_sender" value="" name="sender"></dd>
    <dt><label for="f_cc_myself">Cc myself</label></dt>
    <dd><input type="checkbox" id="f_cc_myself" name="cc_myself"></dd>
  </dl>
  <div class="actions"><input type="submit" value="submit"></div>
</form>
```

Customizing the form template

If the default generated HTML is not to your taste, you can completely customize the way a form is presented using 'call' tag of jinja2. When you use 'call' tag, you need to put your form's contents(including submit buttons) between {% call form() %} and {% endcall %}. Here's an example of how to customize the representation of our form.

```
<body>
{% call form() %}
  <div class="fieldWrapper">
    {{ form.subject.label(class_="myLabel")|safe }}
    {{ form.subject()|safe }}
  </div>
  <div class="fieldWrapper">
    {{ form.message.errors()|safe }}
    {{ form.message.label()|safe }}
    {{ form.message.render()|safe }}
  </div>
  <div class="fieldWrapper">
    {{ form.sender.label()|safe }}
    {{ form.sender.render()|safe }}
    {% if form.message.errors %}
      <span class="errors">
        {% for error in form.message.errors %}
          {{ error }}&nbsp;
        {% endfor %}
      </span>
    {% endif %}
  </div>
  <div class="fieldWrapper">
    {{ form.cc_myself.label()|safe }}
    {{ form.cc_myself.render()|safe }}
    {{ form.cc_myself.errors(class_="myErrors")|safe }}
  </div>
  {{ form.default_actions()|safe }}
{% endcall %}
</body>
```

The example above shows four different ways to display one field widget. You can access each field through the root widget's attribute. Let's take a look in turn.

1. First example

```
<div class="fieldWrapper">
  {{ form.subject.label(class_="myLabel")|safe }}
  {{ form.subject()|safe }}
</div>
```

This code renders the label of the subject field in "myLabel" class. The word "class" is reserved, so you need to add an underscore to avoid error in order to specify the class. The subject field widget is also callable, and if you call it, you can get HTML for both of the input field and error messages at a time.

2. Second example

```
<div class="fieldWrapper">
  {{ form.message.errors()|safe }}
  {{ form.message.label()|safe }}
  {{ form.message.render()|safe }}
```

```
</div>
```

The second example shows you how to separate HTMLs of input field and error messages. If you call render() method instead of just call the field widget, you only get the HTML of input field. So in most cases, you need to put codes for displaying error messages. In this example, you will get this HTML for error messages:

```
<ul class="errors"><li>This field is required.</li></ul>
```

What if you don't like <ul> tags?

3. Third example

```
<div class="fieldWrapper">
  {{ form.sender.label()|safe }}
  {{ form.sender.render()|safe }}
  {% if form.message.errors %}
    <span class="errors">
      {% for error in form.message.errors %}
        {{ error }}&nbsp;
      {% endfor %}
    </span>
  {% endif %}
</div>
```

The third example shows you how to iterate over error messages. Isn't is easy?

4. Forth example

```
<div class="fieldWrapper">
  {{ form.cc_myself.label()|safe }}
  {{ form.cc_myself.render()|safe }}
  {{ form.cc_myself.errors(class_="myErrors")|safe }}
</div>
```

The last example show you how to specify class on error messages. Actually, you can specify any attribute on any renderable widget by passing keyword argument on rendering.

Handling file upload

If your form contains FileField or Field class drived from it, the widget automatically rendered with necessary attribute in its form tag. You need to pass request.files as well as request.form. Here's an example that shows you how to handle file upload.

```
# forms.py
class UploadForm(forms.Form):
  comment = forms.TextField(required=True)
  upload_file = forms.FileField(required=True)

# views.py
form = UploadForm()
if request.method == "POST":
  if form.validate(request.form, request.files):
    # process the data
    # ...
    return redirect("/thanks")
```

Customizing form validation

To put validation method on particular field, you can define a method named 'validate_FIELDNAME'. e.g. To check if a value submitted as 'password' field is stronger enough, you can set 'validate_password' method in the class definition of the Form. If validation fails, you need to raise ValidationError with appropriate error message.

Here's an example:

```
from kay.utils import forms
from kay.utils.validators import ValidationError

class RegisterForm(forms.Form):
  username = forms.TextField(required=True)
  password = forms.TextField(required=True, widget=forms.PasswordInput)

  def validate_password(self, value):
    if not stronger_enough(value):
      raise ValidationError(u"The password you specified is too week.")
```

What if adding a field for password confirmation? To do that, you have to check the values among plural fields, creating the method named 'context_validate'. Here's an example:

```
from kay.utils import forms
from kay.utils.validators import ValidationError

class RegisterForm(forms.Form):
  username = forms.TextField(required=True)
  password = forms.TextField(required=True, widget=forms.PasswordInput)
  password_confirm = forms.TextField(required=True, widget=forms.PasswordInput)

  def validate_password(self, value):
    if not stronger_enough(value):
      raise ValidationError(u"The password you specified is too week.")

  def context_validate(self, data):
    if data['password'] != data['password_confirm']:
      raise ValidationError(u"The passwords don't match.")
```