Managing Forms (for Studio 38, Project Jetson)

# History

|  |  |
| --- | --- |
| Created by | Rupert |
| Created | 28-01-2009 |
| Last modified by | Rupert |
| Last modified | 05-02.2009 |

# Introduction

Currently, the way, Django supports Form processing, is a typical two step mechanism. The first step is: create a form with its fields and validators, and the second step is: write a view to call the form and process the posted data. This document describes a simple mechanism to merge these steps into one (nearly ☺). The idea for that is taken from [1]. We just need to create a form class as we used to do and a second class, which extends a special FormHandler class, put that class at the appropriate place in the urls.py, and all your Form Processing is automated as far as possible without losing the needed flexibility.

# How to Use

Let’s start with an example: We just want to create a very simple form, where you can enter your name. The form should add a new person to a Person model, which we do not describe here.

We start we some url definition in the urls.py:

|  |
| --- |
| (r*'^/person/add/$'*, NameFormHandler(NameForm), {*'action'* : ID\_ACTION\_NEW}), |

The “view” to be called is just an instance of the NameFormHandler class (defined below). Next we write our form:

|  |
| --- |
| class NameForm(forms.Form): |
| name = forms.CharField( |
| label= \_(*"Name"*), |
| required=True, |
| ) |

There is nothing special about that. The third step is setting up an appropriate form template:

|  |
| --- |
| ... |
| <form action="" method="post"> |
| {{ form.title.label }} |
| {{ form.title }} |
| <input type="submit" name="{{ cancel\_stage\_field }}" value="Cancel" /> |
| <input type="submit" name="{{ post\_stage\_field }}" value="Save" /> |
| </form> |
| ... |

There is also nothing special about that template, except the naming of the submit buttons. During form processing, some additional context variables are created, which must be used to define the name of the corresponding submit buttons. These are:

|  |  |
| --- | --- |
| cancel\_stage\_field | name for the “Cancel” submit button |
| post\_stage\_field | name for the “Post” resp. “Save” submit button |

There are some additional context variables for other purposes such as “delete” or “preview”, which are described later in the section.

Last, we write our FormHandler class, which manages the form processing:

|  |
| --- |
| from general.base.forms.formprocessing import FormHandler |
| ... |
| class **NameFormHandler**(FormHandler): |
| form\_template = *'person/forms/name\_form.html'* |
| def **parse\_extra\_params**(*self*, \*args, \*\*kwargs): |
| return {*'title' : 'This is my name form‘*} |
|  |
| def **cancel**(*self*, request, action): |
| return HttpResponseRedirect(*'.'*) |
|  |
| def **done**(*self*, request, action, cleaned): |
| name = cleaned[*'name'*] |
| person = Person(name=name) |
| person.save() |
| return HttpResponseRedirect(*'.'*) |

All methods used are callbacks. They are called by the superclass on specific events or on init. First of all, the form\_template class variable defines the template used to display the form. You must specify the template file. The cancel method and the done method you must override. parse\_extra\_params is optional. A detailed description how to use this class is given in the next section.

# FormHandler classes

Let’s have a closer look at the class instantiation and its methods. All the classes described in this section extend the FormHandler class. Additionally, you must specify the required action, like “new”, “edit” or delete”.

## FormHandler(form, use\_ajax=False)

|  |  |
| --- | --- |
| **Usage** | Can be Instantiated |
| **Purpose** | Used to create a new object using a form or edit an existing an object or delete an object with an optional form to confirm the delete action. |
| **Allowed actions** | New, edit, delete |

All the FormHandler extending classes are instantiated by providing one mandatory and two optional parameter:

|  |  |
| --- | --- |
| form | a form class (not an instance of the form!) |
| use\_ajax | a Boolean parameter, if you want to use ajax forms. Currently, ajax forms are not supported using this mechanism. If not provided, the default value is false. |
| confirm\_delete | If you want to provide a “delete” action and set this parameter to true, a confirmation form “Do you relly want to delete ...” will be displayed before deleting an object. |

In your form templates, you must give the corresponding buttons predefined names, that the form processor can identify the requested submit action correctly. These names are

|  |  |
| --- | --- |
| cancel\_stage\_field | Name for the “Cancel” submit button |
| post\_stage\_field | Name for the “Post” resp. “Save” submit button |
| Delete\_stage\_field | Name for the “Delete” submit button |

In the following section, all method of the FormHandler class are listed. There are some methods, which are delegated from the FormHandler superclass to the concrete subclass you use and which are mandatory. Also, there are some methods, which are delegated from the superclass too, but apply only to specific actions or cases, and last, there are some optional methods you may use but need not to override.

## Methods you must override in any case

### cancel(action)

|  |  |
| --- | --- |
| **Implementation** | Mandatory, must be overridden |
| **Must return** | Some HTTPResponse object or a HTTPRedirect |

cancel is called when pressing the cancel button in the provided form. How to add this button to the form template is described below. You also have access to currently processed action, which has one of the string values “new”, “edit” or “delete”. More on that later. Usually, you will just place some redirection here returning an HTTPRedirect object.

|  |
| --- |
| def **cancel**(*self*, action): |
| return HttpResponseRedirect(*'get\_back'*) |

## Methods you must override depending on the provided actions

### save\_new( cleaned)

|  |  |
| --- | --- |
| **Implementation** | Mandatory, if you provide the “new” action. |
| **Must return** | Some HTTPResponse object or a HTTPRedirect |

is called when pressing the post button in the form, if a “new” action is processed. You get access to the cleaned form data. Usually you will create an instance of a model, initialize the instance with the form data and save the instance. After that, you should redirect to some page.

|  |
| --- |
| def **save\_new**(*self*, cleaned): |
| thread = ForumThread(message=cleaned[*'message'*]) |
| thread.save() |
| return HttpResponseRedirect(*'get\_back'*) |

### save\_edit( object, cleaned)

|  |  |
| --- | --- |
| **Implementation** | Mandatory, if you provide the “edit” action. |
| **Must return** | Some HTTPResponse object or a HTTPRedirect |

is called when pressing the post button in the form, if a “edit” action is processed. You get access to the cleaned form data and the object you are currently editing. Usually you will change the attributes of the provided object and save it. After that, you should redirect to some page.

|  |
| --- |
| def **save\_edit**(*self*, object, cleaned): |
| object.message = cleaned[*'message'*] |
| object.save() |
| return HttpResponseRedirect(*'get\_back'*) |

### delete( object)

|  |  |
| --- | --- |
| **Implementation** | Mandatory, if you provide the “delete” action. |
| **Must return** | Some HTTPResponse object or a HTTPRedirect |

is called, when a delete action is processed. If you set the confirm\_delete flag true on class instantiation, a special delete form is displayed with a delete button. If you press the delete button, the delete()method is called.

|  |
| --- |
| def delete(*self*, object): |
| object.delete() |
| return HttpResponseRedirect(*'get\_back'*) |

### get\_edit\_data(object)

|  |  |
| --- | --- |
| **Implementation** | Mandatory, if you provide the “edit” action. |
| **Must return** | The initial form data as a dictionary |

is called, when the for processing needs the data of the object to edit. You have access to the object you currently edit. The return value must be a dictionary of the form data. A typical example for an implementation would be:

|  |
| --- |
| def **get\_edit\_data**(*self*, object): |
| return {*'message'* : object.message} |

### get\_object()

|  |  |
| --- | --- |
| **Implementation** | Mandatory, if you provide the “edit” or “delete” action. |
| **Must return** | The object to edit or delete |

Provides the object instance, you want to edit or delete.

|  |
| --- |
| def **get\_object**(*self*): |
| return *self*.current\_thread #current thread is set in the parse\_extra\_params method! |

## Methods you may override

### parse\_extra\_params(\*args, \*\*kwargs)

|  |  |
| --- | --- |
| **Implementation** | Optional, may be overridden |
| **Must Return** | A dictionary (or nothing) |

This method is called when the init. \*args and \*\*kwargs hold the parameters passed from the URL (as defined in the urls.py. For example, if you have something like url(r*'example/(?P<some\_url\_part>.\*)/',* …) in your url definition, the some\_url\_part parameter will be into \*\*kwargs. You can use those parameters to calculate extra context for the template or define your own attributes. Mind that you **must not override the** \_\_init\_\_ **method!** Usually, you define a dictionary of context variables to be passed to the form template here.

|  |
| --- |
| def **parse\_extra\_params**(*self*, \*args, \*\*kwargs): |
| extra\_context = get\_forum\_params(\*args, \*\*kwargs) # this collects all needed vars |
| self.current\_thread = extra\_context[*'current\_thread'*]  return extra\_context |

### get\_form\_params()

|  |  |
| --- | --- |
| **Implementation** | Optional, may be overridden |
| **Must Return** | A dictionary (or nothing) |

Maybe, you want to pass some kwargs to your forms custom init method used for validation or anything else in the form. In this case, just override this function. Must return a dictionary.

|  |
| --- |
| def **get\_form\_params**(*self*): |
| return {*'current\_thread'* : *self*.current\_thread} |

### get\_form\_template(use\_ajax)

|  |  |
| --- | --- |
| **Implementation** | Optional, may be overridden |
| **Must Return** | A template name to use for displaying the form. |

Maybe, you want to specify a template for the form dynamically (Alternatively, you can specify the template to use by the form\_template class attribute). Sometimes, you will want to provide different templates for different cases, so just override this method. A typical scenario, when this method is overridden, is described in [2].

|  |
| --- |
| def **get\_form\_template**(*self*, use\_ajax): |
| # a custom function selecting the proper template to use |
| return select\_template(*"thread"*, *self*.thread, use\_ajax) |

### get\_confirm\_delete\_template(use\_ajax)

|  |  |
| --- | --- |
| **Implementation** | Optional, may be overridden |
| **Must Return** | A template name to use for displaying the delete confirmation form. |

The same for the “confirm delete” template (Alternatively, you can specify the template to use by the confirm\_delete\_template class attribute).

|  |
| --- |
| def **get\_confirm\_delete\_template**(*self*, use\_ajax): |
| # a custom function selecting the proper template to use |
| return select\_template(*"confirm\_delete"*, *self*.thread, use\_ajax) |

### check\_allowed(request, action)

|  |  |
| --- | --- |
| **Implementation** | Optional, may be overridden |
| **Must Return** | Some HTTPResponse object or a HTTPRedirect or nothing. |

This method is delegated from the FormHandler superclass, before the form is displayed. Here, some optional checks shall be implemented. If that check fails, this method should return a HttpResponse or HttpResponseRedirect object. It also can also be decorated (by a “login\_required” decorator) or something else. Typically, checks for privileges are placed here.

|  |
| --- |
| def **check\_allowed**(*self*, request, action): |
| ... # Some privilege checking here |
| current\_thread = *self*.current\_thread |
| # check, if you may create a thread here! |
| if action == ID\_ACTION\_NEW: |
| if current\_forum.has\_children(): |
| return HttpResponseRedirect(*'get\_back'*) |
| # the login\_required decorator is also placed here! |
| check\_allowed = login\_required(check\_allowed) |

### check\_warnings(request, action)

|  |  |
| --- | --- |
| **Implementation** | Optional, may be overridden |
| **Must Return** | Some HTTPResponse object or a HTTPRedirect or nothing. |

Also delegated from the FormHandler superclass, before the form is displayed. Here you can place some custom messages to be returned, which should be displayed in the form as “Warnings”, such as “If you create a new forum here, all threads in the existing forum will be lost.”

|  |
| --- |
| def **check\_warnings(***self*, request, action): |
| # check, if you may create a forum here! |
| if action == ID\_ACTION\_NEW: |
| if *self*.current\_forum and *self*.current\_forum.has\_threads(): |
| return \_(*"WARNING! You want to create a new forum …”*) |
| return None |

## FormPreviewHandler(form, use\_ajax=False)

|  |  |
| --- | --- |
| **Usage** | Can be Instantiated |
| **Purpose** | Used to create a new object using a form or edit an existing an object or delete an object with an optional form to confirm the delete action. New and edit actions get an additional preview form. |
| **Allowed actions** | New, edit, delete |

The FormPreviewHandler extends the FormHandler class providing an additional preview form, where you can check your data before posting. The init parameters are the same as for FormHandler extending classes.

The names for the submit buttons in the templates are the same as for the FormHandler class. Additionally, the following button applies to FormPreviewHandler extending classes:

|  |  |
| --- | --- |
| preview\_stage\_field | Name for the “Preview” submit button |

## Methods you must override

The same as for the FormHandler class.

## Methods you may override

Additionally, the following methods may be overridden:

### get\_preview\_template(use\_ajax)

|  |  |
| --- | --- |
| **Implementation** | Optional, may be overridden |
| **Must Return** | A template name to use for displaying the preview form. |

Like the get\_form\_template method, the get\_preview\_template method provides a template name for the preview template dynamically. Alternatively, you can specify the preview template to use by the form\_preview\_template class attribute).

|  |
| --- |
| def **get\_preview\_template**(*self*, use\_ajax): |
| # a custom function selecting the proper template to use |
| return select\_template(*"preview\_thread"*, *self*.thread, use\_ajax) |

There are two additional methods, you may override, which are taken from [1]. For further details, refer to the given reference.

### security\_hash(request, form)

For details, refer to [1].

### failed\_hash(request, action)

For details, refer to [1].

# References

|  |  |
| --- | --- |
|  | „Form Preview”, the Django Documentation, available via <http://docs.djangoproject.com/en/dev/ref/contrib/formtools/form-preview/> |
|  | “Processing Overridable Templates”, available via <https://code.jetsonproject.com/svn/jetson/trunk/jetson/resources/modeling/base/Overridable_template_processing.docx> |