# django-formwizard Documentation

***Release 1.0***

**Stephan Jaekel**

March 06, 2013

# CONTENTS

**Warning:** The 1.0+ releases are incompatible with all previous releases (<=0.6) of django-formwizard!
If you want to use the old version, please install django-formwizard==0.6 (version 0.6 is the last version with the old api)

django-formwizard is a reusable app to work with multi-page forms. Besides normal *Forms*, it supports *FormSets*, *ModelForms* and *ModelFormSets*.

---

**Note:** This app was originally developed as an external library for Django but the code made it to Django itself. Beginning with release 1.4 of Django, the form-wizard will be available in Django directly (*django.contrib.formtools.wizard*).

This code is a backport of the *django.contrib.formtools.wizard* code!

Until the 1.4 release, django-formwizard will be maintained to let Django 1.3 users work with the new form-wizard.

---

To install django-formwizard, simply run

```
# pip install django-formwizard
```

The source is available on GitHub.

If you are interested in contributing, just fork the repository on GitHub and commit your changes. Don't forget to send a pull request.

Contents:

CHAPTER

ONE

# GETTING STARTED

This documentation shows the basic functionality of **django-formwizard**.

Before we start, make sure that you installed **django-formwizard**:

```
pip install django-formwizard
```

django-formwizard provides an "form wizard" application that splits forms across multiple Web pages. It maintains state in one of the backends so that the full server-side processing can be delayed until the submission of the final form.

You might want to use this if you have a lengthy form that would be too unwieldy for display on a single page. The first page might ask the user for core information, the second page might ask for less important information, etc.

The term "wizard", in this context, is explained on Wikipedia.

## 1.1 How it works

Here's the basic workflow for how a user would use a wizard:

1. The user visits the first page of the wizard, fills in the form and submits it.
2. The server validates the data. If it's invalid, the form is displayed again, with error messages. If it's valid, the server saves the current state of the wizard in the backend and redirects to the next step.
3. Step 1 and 2 repeat, for every subsequent form in the wizard.
4. Once the user has submitted all the forms and all the data has been validated, the wizard processes the data – saving it to the database, sending an email, or whatever the application needs to do.

## 1.2 Usage

This application handles as much machinery for you as possible. Generally, you just have to do these things:

1. Define a number of `Form` classes – one per wizard page.
2. Create a `WizardView` subclass that specifies what to do once all of your forms have been submitted and validated. This also lets you override some of the wizard's behavior.
3. Create some templates that render the forms. You can define a single, generic template to handle every one of the forms, or you can define a specific template for each form.
4. Add `formwizard` to your *INSTALLED_APPS* list in your settings file.

5. Point your URLconf at your `WizardView` `as_view()` method.

### 1.2.1 Defining `Form` classes

The first step in creating a form wizard is to create the `Form` classes. These classes can live anywhere in your codebase, but convention is to put them in a file called `forms.py` in your application.

For example, let's write a "contact form" wizard, where the first page's form collects the sender's email address and subject, and the second page collects the message itself. Here's what the `forms.py` might look like:

```
from django import forms

class ContactForm1(forms.Form):
    subject = forms.CharField(max_length=100)
    sender = forms.EmailField()

class ContactForm2(forms.Form):
    message = forms.CharField(widget=forms.Textarea)
```

---

**Note:** In order to use `FileField` in any form, see the section *Handling files* below to learn more about what to do.

---

### 1.2.2 Creating a `WizardView` class

The next step is to create a `formwizard.views.WizardView` subclass. You can also use the `SessionWizardView` or `CookieWizardView` class which preselects the wizard storage backend.

---

**Note:** To use the `SessionWizardView` follow the instructions in the sessions documentation of Django on how to enable sessions.

---

We will use the `SessionWizardView` in all examples but is is completly fine to use the `CookieWizardView` instead. As with your `Form` classes, this `WizardView` class can live anywhere in your codebase, but convention is to put it in `views.py`.

The only requirement on this subclass is that it implement a `done()` method.

`WizardView.`**`done`**(*form_list*)

This method specifies what should happen when the data for *every* form is submitted and validated. This method is passed a list of validated `Form` instances.

In this simplistic example, rather than performing any database operation, the method simply renders a template of the validated data:

```
from django.shortcuts import render_to_response
from formwizard.views import SessionWizardView

class ContactWizard(SessionWizardView):
    def done(self, form_list, **kwargs):
        return render_to_response('done.html', {
            'form_data': [form.cleaned_data for form in form_list],
        })
```

Note that this method will be called via `POST`, so it really ought to be a good Web citizen and redirect after processing the data. Here's another example:

```
from django.http import HttpResponseRedirect
from formwizard.views import SessionWizardView

class ContactWizard(SessionWizardView):
    def done(self, form_list, **kwargs):
        do_something_with_the_form_data(form_list)
        return HttpResponseRedirect('/page-to-redirect-to-when-done/')
```

See the section *Advanced WizardView methods* below to learn about more `WizardView` hooks.

### 1.2.3 Creating templates for the forms

Next, you'll need to create a template that renders the wizard's forms. By default, every form uses a template called `formwizard/wizard_form.html`. You can change this template name by overriding either the `template_name` attribute or the `get_template_names()` method, which is documented below. This hook also allows you to use a different template for each form.

This template expects a `wizard` object that has various items attached to it:

- `form` – The `Form` instance for the current step (either empty or with errors).
- `steps` – A helper object to access the various steps related data:
  - `step0` – The current step (zero-based).
  - `step1` – The current step (one-based).
  - `count` – The total number of steps.
  - `first` – The first step.
  - `last` – The last step.
  - `current` – The current (or first) step.
  - `next` – The next step.
  - `prev` – The previous step.
  - `index` – The index of the current step.
  - `all` – A list of all steps of the wizard.

You can supply additional context variables by using the `get_context_data()` method of your `WizardView` subclass.

Here's a full example template:

```
{% extends "base.html" %}

{% block content %}
<p>Step {{ wizard.steps.current }} of {{ wizard.steps.count }}</p>
<form action="." method="post">{% csrf_token %}
<table>
{{ wizard.management_form }}
{% if wizard.form.forms %}
    {{ wizard.form.management_form }}
    {% for form in wizard.form.forms %}
        {{ form }}
    {% endfor %}
{% else %}
    {{ wizard.form }}
```

```
{% endif %}
{% if wizard.steps.prev %}
<button name="wizard_prev_step" value="{{ wizard.steps.first }}">{% trans "first step" %}</button>
<button name="wizard_prev_step" value="{{ wizard.steps.prev }}">{% trans "prev step" %}</button>
{% endif %}
</table>
<input type="submit">
</form>
{% endblock %}
```

**Note:** Note that `{{ wizard.management_form }}` **must be used** for the wizard to work properly.

### 1.2.4 Hooking the wizard into a URLconf

Finally, we need to specify which forms to use in the wizard, and then deploy the new `WizardView` object a URL in the `urls.py`. The wizard's `as_view()` method takes a list of your `Form` classes as an argument during instantiation:

```
from django.conf.urls.defaults import patterns

from myapp.forms import ContactForm1, ContactForm2
from myapp.views import ContactWizard

urlpatterns = patterns('',
    (r'^contact/$', ContactWizard.as_view([ContactForm1, ContactForm2])),
)
```

## 1.3 Advanced `WizardView` methods

**class `WizardView`**

Aside from the `done()` method, `WizardView` offers a few advanced method hooks that let you customize how your wizard works.

Some of these methods take an argument `step`, which is a zero-based counter as string representing the current step of the wizard. (E.g., the first form is `'0'` and the second form is `'1'`)

`WizardView.`**`get_form_prefix`**(*step*)

Given the step, returns a form prefix to use. By default, this simply uses the step itself. For more, see the *form prefix documentation*.

`WizardView.`**`process_step`**(*form*)

Hook for modifying the wizard's internal state, given a fully validated `Form` object. The Form is guaranteed to have clean, valid data.

Note that this method is called every time a page is rendered for *all* submitted steps.

The default implementation:

```
def process_step(self, form):
    return self.get_form_step_data(form)
```

`WizardView.`**`get_form_initial`**(*step*)

Returns a dictionary which will be passed to the form for `step` as `initial`. If no initial data was provied while initializing the form wizard, a empty dictionary should be returned.

The default implementation:

```
def get_form_initial(self, step):
    return self.initial_dict.get(step, {})
```

WizardView.**get_form_instance**(*step*)
Returns a object which will be passed to the form for `step` as `instance`. If no instance object was provied while initializing the form wizard, None be returned.

The default implementation:

```
def get_form_instance(self, step):
    return self.instance_dict.get(step, None)
```

WizardView.**get_form_kwargs**(*step*)
Returns a dictionary which will be used as the keyword arguments when instantiating the form instance on given `step`.

The default implementation:

```
def get_form_kwargs(self, step):
    return {}
```

WizardView.**get_context_data**(*form*, ***kwargs*)
Returns the template context for a step. You can overwrite this method to add more data for all or some steps. This method returns a dictionary containing the rendered form step.

The default template context variables are:

- Any extra data the storage backend has stored
- `form` – form instance of the current step
- `wizard` – the wizard instance itself

Example to add extra variables for a specific step:

```
def get_context_data(self, form, **kwargs):
    context = super(MyWizard, self).get_context_data(form, **kwargs)
    if self.steps.current == 'my_step_name':
        context.update({'another_var': True})
    return context
```

WizardView.**get_wizard_name**()
This method can be used to change the wizard's internal name.

Default implementation:

```
def get_wizard_name(self):
    return normalize_name(self.__class__.__name__)
```

WizardView.**get_prefix**()
This method returns a prefix for the storage backends. These backends use the prefix to fetch the correct data for the wizard. (Multiple wizards could save their data in one session)

You can change this method to make the wizard data prefix more unique to, e.g. have multiple instances of one wizard in one session.

Default implementation:

```
def get_prefix(self):
    return self.wizard_name
```

WizardView.**get_form**(*step=None*, *data=None*, *files=None*)

This method constructs the form for a given `step`. If no `step` is defined, the current step will be determined automatically. The method gets three arguments:

- `step` – The step for which the form instance should be generated.
- `data` – Gets passed to the form's data argument
- `files` – Gets passed to the form's files argument

You can override this method to add extra arguments to the form instance.

Example code to add a user attribute to the form on step 2:

```
def get_form(self, step=None, data=None, files=None):
    form = super(MyWizard, self).get_form(step, data, files)
    if step == '1':
        form.user = self.request.user
    return form
```

WizardView.**process_step**(*form*)

This method gives you a way to post-process the form data before the data gets stored within the storage backend. By default it just passed the form.data dictionary. You should not manipulate the data here but you can use the data to do some extra work if needed (e.g. set storage extra data).

Default implementation:

```
def process_step(self, form):
    return self.get_form_step_data(form)
```

WizardView.**process_step_files**(*form*)

This method gives you a way to post-process the form files before the files gets stored within the storage backend. By default it just passed the `form.files` dictionary. You should not manipulate the data here but you can use the data to do some extra work if needed (e.g. set storage extra data).

Default implementation:

```
def process_step_files(self, form):
    return self.get_form_step_files(form)
```

WizardView.**render_revalidation_failure**(*step*, *form*, ***kwargs*)

When the wizard thinks, all steps passed it revalidates all forms with the data from the backend storage.

If any of the forms don't validate correctly, this method gets called. This method expects two arguments, `step` and `form`.

The default implementation resets the current step to the first failing form and redirects the user to the invalid form.

Default implementation:

```
def render_revalidation_failure(self, step, form, **kwargs):
    self.storage.current_step = step
    return self.render(form, **kwargs)
```

WizardView.**get_form_step_data**(*form*)

This method fetches the form data from and returns the dictionary. You can use this method to manipulate the values before the data gets stored in the storage backend.

Default implementation:

```
def get_form_step_data(self, form):
    return form.data
```

WizardView.**get_form_step_files**(*form*)

This method returns the form files. You can use this method to manipulate the files before the data gets stored in the storage backend.

Default implementation:

```
def get_form_step_files(self, form):
    return form.files
```

WizardView.**render**(*form*, ***kwargs*)

This method gets called after the get or post request was handled. You can hook in this method to, e.g. change the type of http response.

Default implementation:

```
def render(self, form=None, **kwargs):
    form = form or self.get_form()
    context = self.get_context_data(form, **kwargs)
    return self.render_to_response(context)
```

## 1.4 Providing initial data for the forms

WizardView.**initial_dict**

Initial data for a wizard's `Form` objects can be provided using the optional `initial_dict` keyword argument. This argument should be a dictionary mapping the steps to dictionaries containing the initial data for each step. The dictionary of initial data will be passed along to the constructor of the step's `Form`:

```
>>> from myapp.forms import ContactForm1, ContactForm2
>>> from myapp.views import ContactWizard
>>> initial = {
...     '0': {'subject': 'Hello', 'sender': 'user@example.com'},
...     '1': {'message': 'Hi there!'}
... }
>>> wiz = ContactWizard.as_view([ContactForm1, ContactForm2], initial_dict=initial)
>>> form1 = wiz.get_form('0')
>>> form2 = wiz.get_form('1')
>>> form1.initial
{'sender': 'user@example.com', 'subject': 'Hello'}
>>> form2.initial
{'message': 'Hi there!'}
```

The `initial_dict` can also take a list of dictionaries for a specific step if the step is a `FormSet`.

## 1.5 Handling files

To handle `FileField` within any step form of the wizard, you have to add a `file_storage` to your `WizardView` subclass.

This storage will temporarilyy store the uploaded files for the wizard. The `file_storage` attribute should be a `Storage` subclass.

> **Warning:** Please remember to take care of removing old files as the `WizardView` won't remove any files, whether the wizard gets finished correctly or not.

## 1.6 Conditionally view/skip specific steps

`WizardView.condition_dict`

The `as_view()` accepts a `condition_dict` argument. You can pass a dictionary of boolean values or callables. The key should match the steps name (e.g. '0', '1').

If the value of a specific step is callable it will be called with the `WizardView` instance as the only argument. If the return value is true, the step's form will be used.

This example provides a contact form including a condition. The condition is used to show a message form only if a checkbox in the first step was checked.

The steps are defined in a `forms.py`:

```
from django import forms

class ContactForm1(forms.Form):
    subject = forms.CharField(max_length=100)
    sender = forms.EmailField()
    leave_message = forms.BooleanField(required=False)

class ContactForm2(forms.Form):
    message = forms.CharField(widget=forms.Textarea)
```

We define our wizard in a `views.py`:

```
from django.shortcuts import render_to_response
from formwizard.views import SessionWizardView

def show_message_form_condition(wizard):
    # try to get the cleaned data of step 1
    cleaned_data = wizard.get_cleaned_data_for_step('0') or {}
    # check if the field ``leave_message`` was checked.
    return cleaned_data.get('leave_message', True)

class ContactWizard(SessionWizardView):

    def done(self, form_list, **kwargs):
        return render_to_response('done.html', {
            'form_data': [form.cleaned_data for form in form_list],
        })
```

We need to add the `ContactWizard` to our `urls.py` file:

```
from django.conf.urls.defaults import pattern

from myapp.forms import ContactForm1, ContactForm2
from myapp.views import ContactWizard, show_message_form_condition

contact_forms = [ContactForm1, ContactForm2]

urlpatterns = patterns('',
    (r'^contact/$', ContactWizard.as_view(contact_forms,
        condition_dict={'1': show_message_form_condition}
    )),
)
```

As you can see, we defined a `show_message_form_condition` next to our `WizardView` subclass and added a `condition_dict` argument to the `as_view()` method. The key refers to the second wizard step (because of

the zero based step index).

## 1.7 How to work with ModelForm and ModelFormSet

The WizardView supports `ModelForm` and `ModelFormSet`. Additionally to the `initial_dict`, the `as_view()` method takes a `instance_dict` argument with a list of instances for the `ModelForm` and `ModelFormSet`.

## 1.8 Usage of NamedUrlWizardView

**class NamedUrlWizardView**

There is a `WizardView` subclass which adds named-urls support to the wizard. By doing this, you can have single urls for every step.

To use the named urls, you have to change the `urls.py`.

Below you will see an example of a contact wizard with two steps, step 1 with "contactdata" as its name and step 2 with "leavemessage" as its name.

Additionally you have to pass two more arguments to the `as_view()` method:

- `url_name` – the name of the url (as provided in the urls.py)
- `done_step_name` – the name in the url for the done step

Example code for the changed `urls.py` file:

```
from django.conf.urls.defaults import url, patterns

from myapp.forms import ContactForm1, ContactForm2
from myapp.views import ContactWizard

named_contact_forms = (
    ('contactdata', ContactForm1),
    ('leavemessage', ContactForm2),
)

contact_wizard = ContactWizard.as_view(named_contact_forms,
    url_name='contact_step', done_step_name='finished')

urlpatterns = patterns('',
    url(r'^contact/(?P<step>.+)/$', contact_wizard, name='contact_step'),
    url(r'^contact/$', contact_wizard, name='contact'),
)
```

CHAPTER

TWO

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*