django-formtools Documentation

Release 2.2

Django Software Foundation and individual contributors

Nov 15, 2020

Contents

django-formtools is a collection of assorted utilities that are useful for specific form use cases.

Currently there are two tools: a helper for form previews and a form wizard view.

CHAPTER 1

Form preview

Django comes with an optional "form preview" application that helps automate the following workflow:

> "Display an HTML form, force a preview, then do something with the submission."

To force a preview of a form submission, all you have to do is write a short Python class.

1.1 Overview

Given a `Form` subclass that you define, this application takes care of the following workflow:

1. Displays the form as HTML on a Web page.
2. Validates the form data when it's submitted via POST. a. If it's valid, displays a preview page. b. If it's not valid, redisplays the form with error messages.
3. When the "confirmation" form is submitted from the preview page, calls a hook that you define – a *`done()`* method that gets passed the valid data.

The framework enforces the required preview by passing a shared-secret hash to the preview page via hidden form fields. If somebody tweaks the form parameters on the preview page, the form submission will fail the hash-comparison test.

1.2 How to use `FormPreview`

1. Point Django at the default FormPreview templates. There are two ways to do this:
   - Add `'formtools'` to your `INSTALLED_APPS` setting.

     This will work if your `TEMPLATES` setting includes the `app_directories` template loader (which is the case by default).

     See the template loader docs for more.

- Otherwise, determine the full filesystem path to the `formtools/templates` directory and add that directory to your `DIRS` option in the `TEMPLATES` setting.

2. Create a *`FormPreview`* subclass that overrides the `done()` method:

```
from django.http import HttpResponseRedirect

from formtools.preview import FormPreview
from myapp.models import SomeModel

class SomeModelFormPreview(FormPreview):

    def done(self, request, cleaned_data):
        # Do something with the cleaned_data, then redirect
        # to a "success" page.
        return HttpResponseRedirect('/form/success')
```

   This method takes an `HttpRequest` object and a dictionary of the form data after it has been validated and cleaned. It should return an `HttpResponseRedirect` that is the end result of the form being submitted.

3. Change your URLconf to point to an instance of your *`FormPreview`* subclass:

```
from django import forms

from myapp.forms import SomeModelForm
from myapp.preview import SomeModelFormPreview
```

   ...and add the following line to the appropriate model in your URLconf:

```
path('post/', SomeModelFormPreview(SomeModelForm)),
```

   where `SomeModelForm` is a Form or ModelForm class for the model.

4. Run the Django server and visit `/post/` in your browser.

1.3 `FormPreview` classes

class formtools.preview.**FormPreview**

A *`FormPreview`* class is a simple Python class that represents the preview workflow. *`FormPreview`* classes must subclass `FormPreview` and override the `done()` method. They can live anywhere in your codebase.

1.4 `FormPreview` templates

FormPreview.**form_template**

FormPreview.**preview_template**

By default, the form is rendered via the template `formtools/form.html`, and the preview page is rendered via the template `formtools/preview.html`.

These values can be overridden for a particular form preview by setting *`preview_template`* and *`form_template`* attributes on the FormPreview subclass. See `formtools/templates` for the default templates.

1.5 Required methods

`FormPreview.`**`done`**(*request*, *cleaned_data*)
Does something with the `cleaned_data` data and then needs to return an `HttpResponseRedirect`, e.g. to a success page.

1.6 Optional methods

`FormPreview.`**`get_auto_id`**()
Hook to override the `auto_id` kwarg for the form. Needed when rendering two form previews in the same template.

`FormPreview.`**`get_initial`**(*request*)
Takes a request argument and returns a dictionary to pass to the form's `initial` kwarg when the form is being created from an HTTP get.

`FormPreview.`**`get_context`**(*request*, *form*)
Context for template rendering.

`FormPreview.`**`parse_params`**(*request*, **args*, ***kwargs*)
Given captured args and kwargs from the URLconf, saves something in self.state and/or raises `Http404` if necessary.

For example, this URLconf captures a user_id variable:

```
path('contact/<int:user_id>/', MyFormPreview(MyForm)),
```

In this case, the kwargs variable in parse_params would be `{'user_id': 32}` for a request to `'/contact/32/'`. You can use that `user_id` to make sure it's a valid user and/or save it for later, for use in *`done()`*.

`FormPreview.`**`process_preview`**(*request*, *form*, *context*)
Given a validated form, performs any extra processing before displaying the preview page, and saves any extra data in context.

By default, this method is empty. It is called after the form is validated, but before the context is modified with hash information and rendered.

`FormPreview.`**`security_hash`**(*request*, *form*)
Calculates the security hash for the given `HttpRequest` and `Form` instances.

Subclasses may want to take into account request-specific information, such as the IP address.

`FormPreview.`**`failed_hash`**(*request*)
Returns an `HttpResponse` in the case of an invalid security hash.

CHAPTER 2

Form wizard

The form wizard application 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.

2.1 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.

2.2 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 `formtools` to your `INSTALLED_APPS` list in your settings file.
5. Point your URLconf at your *`WizardView.as_view()`* method.

2.2.1 Defining `Form` classes

The first step in creating a form wizard is to create the `Form` classes. These should be standard `django.forms.Form` classes, covered in the `forms documentation`. 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.

2.2.2 Creating a `WizardView` subclass

class formtools.wizard.views.**SessionWizardView**

class formtools.wizard.views.**CookieWizardView**

The next step is to create a *`formtools.wizard.views.WizardView`* subclass. You can also use the *`SessionWizardView`* or *`CookieWizardView`* classes which preselect the backend used for storing information during execution of the wizard (as their names indicate, server-side sessions and browser cookies respectively).

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

We will use the *`SessionWizardView`* in all examples but is completely 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*, *form_dict*, ***kwargs*)

This method specifies what should happen when the data for *every* form is submitted and validated. This method is passed a list and dictionary 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
from formtools.wizard.views import SessionWizardView

```

(continues on next page)

(continued from previous page)

```
class ContactWizard(SessionWizardView):
    def done(self, form_list, **kwargs):
        return render(self.request, '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 formtools.wizard.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/')
```

In addition to `form_list`, the *`done()`* method is passed a `form_dict`, which allows you to access the wizard's forms based on their step names. This is especially useful when using *`NamedUrlWizardView`*, for example:

```
def done(self, form_list, form_dict, **kwargs):
    user = form_dict['user'].save()
    credit_card = form_dict['credit_card'].save()
    # ...
```

Changed in version 1.7: Previously, the `form_dict` argument wasn't passed to the `done` method.

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

2.2.3 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 `formtools/wizard/wizard_form.html`. You can change this template name by overriding either the `template_name` attribute or the `get_template_names()` method, which are documented in the `TemplateResponseMixin` documentation. The latter one allows you to use a different template for each form (*see the example below*).

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

- `form` – The `Form` or `BaseFormSet` 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" %}
{% load i18n %}

{% block head %}
{{ wizard.form.media }}
{% endblock %}

{% block content %}
<p>Step {{ wizard.steps.step1 }} 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 %}
</table>
{% if wizard.steps.prev %}
<button name="wizard_goto_step" type="submit" value="{{ wizard.steps.first }}">{%␣
↪trans "first step" %}</button>
<button name="wizard_goto_step" type="submit" value="{{ wizard.steps.prev }}">{%␣
↪trans "prev step" %}</button>
{% endif %}
<input type="submit" value="{% trans "submit" %}"/>
</form>
{% endblock %}
```

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

2.2.4 Hooking the wizard into a URLconf

`WizardView.`**`as_view`**`()`

Finally, we need to specify which forms to use in the wizard, and then deploy the new *`WizardView`* object at 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.path import path

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

urlpatterns = [
```

(continues on next page)

(continued from previous page)

```
    path('contact/', ContactWizard.as_view([ContactForm1, ContactForm2])),
]
```

You can also pass the form list as a class attribute named `form_list`:

```
class ContactWizard(WizardView):
    form_list = [ContactForm1, ContactForm2]
```

2.2.5 Using a different template for each form

As mentioned above, you may specify a different template for each form. Consider an example using a form wizard to implement a multi-step checkout process for an online store. In the first step, the user specifies a billing and shipping address. In the second step, the user chooses payment type. If they chose to pay by credit card, they will enter credit card information in the next step. In the final step, they will confirm the purchase.

Here's what the view code might look like:

```
from django.http import HttpResponseRedirect
from formtools.wizard.views import SessionWizardView

FORMS = [("address", myapp.forms.AddressForm),
         ("paytype", myapp.forms.PaymentChoiceForm),
         ("cc", myapp.forms.CreditCardForm),
         ("confirmation", myapp.forms.OrderForm)]

TEMPLATES = {"address": "checkout/billingaddress.html",
             "paytype": "checkout/paymentmethod.html",
             "cc": "checkout/creditcard.html",
             "confirmation": "checkout/confirmation.html"}

def pay_by_credit_card(wizard):
    """Return true if user opts to pay by credit card"""
    # Get cleaned data from payment step
    cleaned_data = wizard.get_cleaned_data_for_step('paytype') or {'method': 'none'}
    # Return true if the user selected credit card
    return cleaned_data['method'] == 'cc'


class OrderWizard(SessionWizardView):
    def get_template_names(self):
        return [TEMPLATES[self.steps.current]]

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

The `urls.py` file would contain something like:

```
urlpatterns = [
    path('checkout/', OrderWizard.as_view(FORMS, condition_dict={'cc': pay_by_credit_
↪card})),
]
```

The `condition_dict` can be passed as attribute for the `as_view()` method or as a class attribute named `condition_dict`:

```
class OrderWizard(WizardView):
    condition_dict = {'cc': pay_by_credit_card}
```

Note that the `OrderWizard` object is initialized with a list of pairs. The first element in the pair is a string that corresponds to the name of the step and the second is the form class.

In this example, the `get_template_names()` method returns a list containing a single template, which is selected based on the name of the current step.

2.3 Advanced `WizardView` methods

class formtools.wizard.views.**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=None*, *form=None*)

Returns the prefix which will be used when calling the form for the given step. `step` contains the step name, `form` the form class which will be called with the returned prefix.

If no `step` is given, it will be determined automatically. By default, this simply uses the step itself and the `form` parameter is not used.

For more, see the form prefix documentation.

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

Returns a dictionary which will be passed as the `initial` argument when instantiating the Form instance for step `step`. If no initial data was provided while initializing the form wizard, an empty dictionary should be returned.

The default implementation:

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

`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_form_instance`**(*step*)

This method will be called only if a `ModelForm` is used as the form for step `step`.

Returns an `Model` object which will be passed as the `instance` argument when instantiating the `ModelForm` for step `step`. If no instance object was provided while initializing the form wizard, `None` will be returned.

The default implementation:

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

`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
- `wizard` – a dictionary representation of the wizard instance with the following key/values:
  - `form` – `Form` or `BaseFormSet` instance for the current step
  - `steps` – A helper object to access the various steps related data
  - `management_form` – all the management data for the current step

Example to add extra variables for a specific step:

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

`WizardView.`**`get_prefix`**(*request*, **args*, ***kwargs*)

This method returns a prefix for use by the storage backends. Backends use the prefix as a mechanism to allow data to be stored separately for each wizard. This allows wizards to store their data in a single backend without overwriting each other.

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, request, *args, **kwargs):
    # use the lowercase underscore version of the class name
    return normalize_name(self.__class__.__name__)
```

Changed in version 1.0: The `request` parameter was added.

`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. If you override `get_form`, however, you will need to set `step` yourself using `self.steps.current` as in the example below. 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().get_form(step, data, files)

    # determine the step if not given
    if step is None:
        step = self.steps.current
```

(continues on next page)

(continued from previous page)

```
    if step == '1':
        form.user = self.request.user
    return form
```

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.

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 returns the `form.data` dictionary. You should not manipulate the data here but you can use it to do some extra work if needed (e.g. set storage extra 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.**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 returns the `form.files` dictionary. You should not manipulate the data here but you can use it 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_goto_step**(*step*, *goto_step*, ***kwargs*)

This method is called when the step should be changed to something else than the next step. By default, this method just stores the requested step `goto_step` in the storage and then renders the new step.

If you want to store the entered data of the current step before rendering the next step, you can overwrite this method.

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

When the wizard thinks all steps have 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 data from the `form` Form instance 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 has been 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=form, **kwargs)
    return self.render_to_response(context)
```

WizardView.**get_cleaned_data_for_step**(*step*)

This method returns the cleaned data for a given `step`. Before returning the cleaned data, the stored values are revalidated through the form. If the data doesn't validate, `None` will be returned.

WizardView.**get_all_cleaned_data**()

This method returns a merged dictionary of all form steps' `cleaned_data` dictionaries. If a step contains a `FormSet`, the key will be prefixed with `formset-` and contain a list of the formset's `cleaned_data` dictionaries. Note that if two or more steps have a field with the same name, the value for that field from the latest step will overwrite the value from any earlier steps.

2.4 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!'}
... }
>>> # This example is illustrative only and isn't meant to be run in
>>> # the shell since it requires an HttpRequest to pass to the view.
>>> wiz = ContactWizard.as_view([ContactForm1, ContactForm2], initial_
↪dict=initial)(request)
>>> 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`.

The `initial_dict` can also be added as a class attribute named `initial_dict` to avoid having the initial data in the `urls.py`.

2.5 Handling files

`WizardView.`**`file_storage`**

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

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

Django provides a built-in storage class (see the built-in filesystem storage class):

```
from django.conf import settings
from django.core.files.storage import FileSystemStorage

class CustomWizardView(WizardView):
    ...
    file_storage = FileSystemStorage(location=os.path.join(settings.MEDIA_ROOT,
↪'photos'))
```

Warning: Please remember to take care of removing old temporary files, as the *`WizardView`* will only remove these files if the wizard finishes correctly.

2.6 Conditionally view/skip specific steps

`WizardView.`**`condition_dict`**

The *`as_view()`* method accepts a `condition_dict` argument. You can pass a dictionary of boolean values or callables. The key should match the steps names (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` file:

```
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
from formtools.wizard.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(self.request, '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.urls import path

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

contact_forms = [ContactForm1, ContactForm2]

urlpatterns = [
    path('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).

2.7 How to work with ModelForm and ModelFormSet

`WizardView.`**`instance_dict`**

WizardView supports `ModelForms` and ModelFormSets. Additionally to *`initial_dict`*, the *`as_view()`* method takes an `instance_dict` argument that should contain model instances for steps based on `ModelForm` and querysets for steps based on `ModelFormSet`.

2.8 Usage of `NamedUrlWizardView`

class `formtools.wizard.views.`**`NamedUrlWizardView`**

class `formtools.wizard.views.`**`NamedUrlSessionWizardView`**

class `formtools.wizard.views.`**`NamedUrlCookieWizardView`**

`NamedUrlWizardView` is a *`WizardView`* subclass which adds named-urls support to the wizard. This allows you to have separate URLs for every step. You can also use the *`NamedUrlSessionWizardView`* or *`NamedUrlCookieWizardView`* classes which preselect the backend used for storing information (Django sessions and browser cookies respectively).

To use the named URLs, you should not only use the *NamedUrlWizardView* instead of *WizardView*, but you will also have to change your `urls.py`.

The *as_view()* method takes two additional arguments:

- a required `url_name` – the name of the url (as provided in the `urls.py`)
- an optional `done_step_name` – the name of the done step, to be used in the URL

This is an example of a `urls.py` for a contact wizard with two steps, step 1 named `contactdata` and step 2 named `leavemessage`:

```
from django.urls import path, re_path

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 = [
    re_path(r'^contact/(?P<step>.+)/$', contact_wizard, name='contact_step'),
    path('contact/', contact_wizard, name='contact'),
]
```

2.9 Advanced `NamedUrlWizardView` methods

`NamedUrlWizardView.`**`get_step_url`**`(`*step*`)`

This method returns the URL for a specific step.

Default implementation:

```
def get_step_url(self, step):
    return reverse(self.url_name, kwargs={'step': step})
```

CHAPTER 3

Changelog

This page details the changes in the various `django-formtools` releases.

3.1 Next Release (TBC)

- Dropped testing for Django 1.11, 2.0 and 2.1.
- Added support for Django 3.1 and Python 3.9.

3.2 2.2 (2019-12-05)

- Dropped testing for Django 1.8, 1.9, 1.10.
- Dropped support for Python 2.
- Added support for Django 2.1, 2.2, 3.0, and Python 3.7.
- Updated translations from Transifex.

3.3 2.1 (2017-10-04)

- Added testing for Django 1.11 (no code changes were required).
- Added support for Django 2.0.
- Dropped testing for Python 3.3 (now end-of-life) on Django 1.8.

3.4 2.0 (2017-01-07)

- Added the `request` parameter to *`FormPreview.parse_params()`*.
- Added support for Django 1.10.
- Dropped support for Django 1.7 and Python 3.2 on Django 1.8.

3.5 1.0 (2015-03-25)

- Added the `request` parameter to *`WizardView.get_prefix()`*.

  This was originally reported and fixed in the main Django repository:

  https://code.djangoproject.com/ticket/19981

- A *form wizard* using the *`CookieWizardView`* will now ignore an invalid cookie, and the wizard will restart from the first step. An invalid cookie can occur in cases of intentional manipulation, but also after a secret key change. Previously, this would raise `WizardViewCookieModified`, a `SuspiciousOperation`, causing an exception for any user with an invalid cookie upon every request to the wizard, until the cookie is removed.

  This was originally reported and fixed in the main Django repository:

  https://code.djangoproject.com/ticket/22638

- Added missing form element to default wizard form template `formtools/wizard/wizard_form.html`.

CHAPTER 4

Installation

To install django-formtools use your favorite packaging tool, e.g.pip:

```
pip install django-formtools
```

Or download the source distribution from PyPI at https://pypi.python.org/pypi/django-formtools, decompress the file and run `python setup.py install` in the unpacked directory.

Then add `'formtools'` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    # ...
    'formtools',
)
```

Note: Adding `'formtools'` to your `INSTALLED_APPS` setting is required for translations and templates to work. Using django-formtools without adding it to your `INSTALLED_APPS` setting is not recommended.

CHAPTER 5

Internationalization

Formtools has its own catalog of translations, in the directory `formtools/locale`, and it's not loaded automatically like Django's general catalog in `django/conf/locale`. If you want formtools's texts to be translated, like the templates, you must include *`formtools`* in the `INSTALLED_APPS` setting, so the internationalization system can find the catalog, as explained in How Django discovers translations.

CHAPTER 6

Releases

New releases of django-formtools should always be compatible with the latest stable release of Django. If a new version of Django contains backwards incompatible changes that affect formtools, a new release of formtools will be issued shortly after the release of the new Django version. Version numbers follow the appropriate Python standards, e.g. PEPs 386 and 440.

CHAPTER 7

How to migrate

If you've used the old `django.contrib.formtools` package follow these two easy steps to update your code:

1. Install the third-party `django-formtools` package.
2. Change your app's import statements to reference the new packages.

   For example, change this:

   ```
   from django.contrib.formtools.wizard.views import WizardView
   ```

   . . . to this:

   ```
   from formtools.wizard.views import WizardView
   ```

The code in the new package is the same (it was copied directly from Django), so you don't have to worry about backwards compatibility in terms of functionality. Only the imports have changed.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

P

R

S

W