Open Source Frameworks for Rapid Application Development

Marek Krętowski
Krzysztof Bandurski, Tomasz Łukaszuk, Tomasz Rybak

Software Departament
Faculty of Computer Science
Bialystok University of Technology

m.kretowski@pb.edu.pl
k.bandurski@pb.edu.pl, t.lukaszuk@pb.edu.pl, t.rybak@pb.edu.pl

Lecture topic

Django forms

Django forms: Table of content

Overview

`django.forms` - Django's form handling library

`django.forms` allows to:

1. Display an HTML form with automatically generated form widgets.
2. Check submitted data against a set of validation rules.
3. Redisplay a form in the case of validation errors.
4. Convert submitted form data to the relevant Python data types.

4 basic concepts

`django.forms` allows to:

1. **Widget**
   A class that corresponds to an HTML form widget, e.g. `<input type="text">` or `<textarea>`. This handles rendering of the widget as HTML.
2. **Field**
   A class that is responsible for doing validation, e.g. an EmailField that makes sure its data is a valid e-mail address.
3. **Form**
   A collection of fields that knows how to validate itself and display itself as HTML.
4. **Form Media**
   The CSS and JavaScript resources that are required to render a form.

Form objects

A Form object encapsulates a sequence of form fields and a collection of validation rules that must be fulfilled in order for the form to be accepted.

```
from django import forms

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

Using forms in a view

```
def contact(request):
    if request.method == 'POST': # If the form has been submitted...
        form = ContactForm(request.POST) # A form bound to the POST data
        if form.is_valid(): # All validation rules pass
            # Process the data in form.cleaned_data
            # ...
            return HttpResponseRedirect('/thanks/') # Redirect after POST
    else:
        form = ContactForm() # An unbound form

    return render_to_response('contact.html', {'form': form,})
```

Three code paths:

1. Create an unbound instance of ContactForm (GET) and pass it to the template
2. Create a bound instance of ContactForm (POST), validate it, redirect to a “thanks” page on successful validation
3. If the form is invalid, pass a bound instance to the template

Processing the data from a form

```
    if form.is_valid():
        subject = form.cleaned_data['subject']
        message = form.cleaned_data['message']
        sender = form.cleaned_data['sender']
        cc_myself = form.cleaned_data['cc_myself']

        recipients = ['info@example.com']
        if cc_myself:
            recipients.append(sender)

        from django.core.mail import send_mail
        send_mail(subject, message, sender, recipients)
        return HttpResponseRedirect('/thanks/') # Redirect after POST
```

`is_valid()` not only verifies the data, but also converts it into relevant Python types.

Three types of validation

1. The `clean()` method on a Field subclass. Returns clean data.
2. The `clean_<fieldname>()` method on a form subclass. Must look into `self.cleaned_data` (containing Python objects). Returns clean data.
3. The `clean()` method in the Form subclass. Can perform validation on multiple fields at once. Returns the final `clean_data`.

Any of these methods can raise `ValidationError`, which will add an item to the relevant errorlist.

Displaying a form using a template

Using `as_p`:

```
<form action="/contact/" method="post">
{{ form.as_p }}
<input type="submit" value="Submit" />
</form>
```

HTML output:

```
<form action="/contact/" method="post">
<p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label>
    <input type="text" name="message" id="id_message" /></p>
<p><label for="id_sender">Sender:</label>
    <input type="text" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
<input type="submit" value="Submit" />
</form>
```

`as_table` would display the form as a table

Customizing the form template

Each named form-field can be output to the template using `{{ form.name_of_field }}`, which will produce the HTML needed to display the form widget.

```
<form action="/contact/" method="post">
    <div class="fieldWrapper">
        {{ form.subject.errors }}
        <label for="id_subject">E-mail subject:</label>
        {{ form.subject }}
    </div>
    <div class="fieldWrapper">
        {{ form.message.errors }}
        <label for="id_message">Your message:</label>
        {{ form.message }}
    </div>
...
    <p><input type="submit" value="Send message" /></p>
</form>
```

- `{{ form.name_of_field }}` produces the widget's HTML
- `{{ form.name_of_field.errors }}` displays a list of for errors

Forms for models

ModelForm

The `ModelForm` class subclasses `Form` and allows to create a form mapped to a Django model.

```
>>> from django.forms import ModelForm

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

Each model field will be represented by a default model field (the full list of conversions is available in the Django docs)

The `save()` method

- Creates and saves a database object from the data bound to the form
- Accepts the `instance` keyword argument - if supplied, updates the given instance instead of creating a new one

```
# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(instance=a)
>>> f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

The `save()` method

- Accepts the `commit` keyword argument - if `True` (default), the model instance is saved to database; if `False`, only creates an instance
- If `commit=False`, any ManyToMany cannot be saved - the instance does not exist in database yet! Use `save_m2m()` afterwards.

```
# Create a form instance with POST data.
>>> f = AuthorForm(request.POST)

# Create, but don't save the new author instance.
>>> new_author = f.save(commit=False)

# Modify the author in some way.
>>> new_author.some_field = 'some_value'

# Save the new instance.
>>> new_author.save()

# Now, save the many-to-many data for the form.
>>> f.save_m2m()
```

Using a subset of fields on the form

- Set `editable=False` on the model field to exclude it from every forms created from the model via ModelForm
- Use the `fields` attribute of the `ModelForm`'s inner `Meta` class (this also allows to change the order of the fiels)
- Use the `exclude` attribute of the `ModelForm`'s inner `Meta` class

```
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title')

class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        exclude = ('birth_date',)
```

Overriding the default field types

- Just declare fields that will override the default ones.

```
>>> class ArticleForm(ModelForm):
...     pub_date = MyDateFormField()
...
...     class Meta:
...         model = Article
```

- To override the widget, just specify the `widget` parameter when declaring the field

```
>>> class ArticleForm(ModelForm):
...     pub_date = DateField(widget=MyDateWidget())
...
...     class Meta:
...         model = Article
```

Form Media

Making forms prettier

- Making an attractive and easy-to-use form requires more than just HTML: CSS stylesheets and JavaScript
- The combination of CSS and JavaScript will depend on particualr widgets used on the page
- Django allows to associate different media files with forms and widgets
- Django Admin defines a number of customized widgets, see: `django.contrib.admin.widgets`
- Django can integrate with any JavaScript toolkit

Static media definitions

The inner `Media` class can define the media requirements.

```
class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')
```

The static media definition are converted at runtime into a widget property by named `media`.

```
>>> w = CalendarWidget()
>>> print w.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="
<script type="text/javascript" src="http://media.example.com/animations.js">
</script>
<script type="text/javascript" src="http://media.example.com/actions.js">
</script>
```

`Media` options

`css`: a dictionary describing the CSS files required for various forms of output media. The values should be tuples/lists of filenames.

- The values should be tuples/lists of filenames.
- The keys should are output media types: all, aural, braille...
- A key can be a comma separated list of various output media types.

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'tv,projector': ('lo_res.css',),
        'print': ('newspaper.css',)
    }
```

`js`: a tuple describing the required JavaScript files.
NOTE: Paths to media files will be prepended with `settings.MEDIA_URL`.

Media on forms

Forms can also have media definitions that are combined with media defined for the form's widgets:

```
    class ContactForm(forms.Form):
        date = DateField(widget=CalendarWidget)
        name = CharField(max_length=40, widget=OtherWidget)

        class Media:
            css = {
                'all': ('layout.css',)
            }
    >>> f = ContactForm()
    >>> f.media
    <link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="
    <link href="http://media.example.com/layout.css" type="text/css" media="all" rel="
    <script type="text/javascript" src="http://media.example.com/animations.js"></scri
    <script type="text/javascript" src="http://media.example.com/actions.js"></script>
    <script type="text/javascript" src="http://media.example.com/whizbang.js"></script
```

Formsets

Purpose

A formset is a layer of abstraction to working with multiple forms on the same page.
If we have a form for creating articles. . .

```
>>> from django import forms
>>> class ArticleForm(forms.Form):
...     title = forms.CharField()
...     pub_date = forms.DateField()
```

. . . we might want to allow users create multiple articles at once:

```
>>> from django.forms.formsets import formset_factory
>>> ArticleFormSet = formset_factory(ArticleForm)
```

We can control the number of extra forms and the maximum number of all forms with `extra` and `max_num` keyword arguments:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_num=1)
```

Using formsets in views and templates

Same as a regular `Form` class. The only difference is that the management form has to be used in the template.

```
def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    if request.method == 'POST':
        formset = ArticleFormSet(request.POST, request.FILES)
        if formset.is_valid():
            # do something with the formset.cleaned_data
    else:
        formset = ArticleFormSet()
    return render_to_response('manage_articles.html', {'formset': formset})
```

`manage_articles.html`:

```
<form method="post" action="">
    {{ formset.management_form }}
    <table>
        {% for form in formset.forms %}
        {{ form }}
        {% endfor %}
    </table>
</form>
```

Model formsets

Created using `modelformset_factory`, which is an extension of `formset_facotry`:

```
>>> from django.forms.models import modelformset_factory
>>> AuthorFormSet = modelformset_factory(Author)
```

Use the `queryset` keyword parameter to limit the number of objects:

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(...))
```

Formsets also have a `save()` method that returns a list of instances saved to the database:

```
# Create a formset instance with POST data.
>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.
>>> instances = formset.save()
```

Basic file uploads

Simple form containing a `FileField`

```
from django import forms

class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file  = forms.FileField()
```

A view handling this form will receive the file data in `request.FILES`, which is a dictionary containing a key for each `FileField` (or `ImageField`, or other `FileField` subclass) in the form. So the data from the above form would be accessible as `request.FILES['file']`.

Basic file uploads (cd)

Note that `request.FILES` will only contain data if the request method was `POST` and the `<form>` that posted the request has the attribute `enctype="multipart/form-data"`. Otherwise, `request.FILES` will be empty.

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render_to_response('upload.html', {'form': form})
```

Handling uploaded files

```
def handle_uploaded_file(f):
    with open('some/file/name.txt', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

class `UploadedFile`

methods/attributes: `read()`, `multiple_chunks()`, `chunks()`, `name`, `size`, `content_type`, `charset`, `temporary_file_path`

Settings control Django's file upload behavior

- FILE_UPLOAD_MAX_MEMORY_SIZE - default 2,5 MB
- FILE_UPLOAD_TEMP_DIR - default tmp
- FILE_UPLOAD_PERMISSIONS
- FILE_UPLOAD_HANDLERS

References

- Django documentation: http://www.djangoproject.com/
- Djangobook: http://www.djangobook.com/