LESSON 8
Forms and Widgets

Topics Covered

☑ Form processing.

☑ Form fields.

☑ Widgets.

☑ Validators.

☑ Crispy Forms.

Introduction

In this lesson, you will create a job application for the Django Jokes website. In doing so, you will learn to work with Django forms and a third-party library for styling forms.

8.1. Form Processing

In Django, form processing usually works like this:

1. User visits page with form. This is a `GET` request.
2. User fills out and submits form. This is a `POST` request.
3. If there are any errors, the form is returned with the form fields still filled in with one or more error messages showing.
4. When the form submission passes validation, the data is processed, and the user is redirected to a success page.

❖ 8.1.1. `ProcessFormView`

Most Django form views (e.g., `FormView`, `CreateView`, `UpdateView`) inherit from `django.views.generic.ProcessFormView`, which includes `get()` and `post()` methods:

- The `get()` method handles the `GET` request, which outputs the form to submit.
- The `post()` method handles the `POST` request, which takes care of processing the form data. Part of that involves checking if the form is valid. When the form is deemed valid, the `form_valid()` method is called. By default, it just redirects to the success page; however, as you shall soon see, it can be overridden to perform additional tasks (e.g., sending an email or saving data to a database) before redirecting to the success page.

8.2. Understanding Form Fields

The term "field" can be confusing in Django development as it can mean several different things:

1. Model Fields (`django.db.models.fields`) – These fields make up a model:

Demo 8.1: jokes/models.py

```
      -------Lines 1 through 5 Omitted-------
6.    class Joke(models.Model):
7.        question = models.TextField(max_length=200)
8.        answer = models.TextField(max_length=100, blank=True)
9.        slug = models.SlugField(
10.           max_length=50, unique=True, null=False, editable=False
11.       )
12.       created = models.DateTimeField(auto_now_add=True)
13.       updated = models.DateTimeField(auto_now=True)
      -------Lines 14 through 25 Omitted-------
```

   We will dig into model fields in the ModelForms lesson (see page 261).

2. Form Fields (`django.forms.fields`) – These fields make up a form. We will cover them in detail in this lesson.
3. HTML Form Fields – These are the HTML form controls that make up an HTML form. In the Django documentation, these are referred to as *HTML input elements*, but that's a little misleading, as not all HTML form controls are `input` elements (e.g., `textarea`, `select`, `button`, etc.). We will use the term *form control* when referring to HTML form fields.

❖ 8.2.1. Default Widgets

Every type of form field has a default *widget*, which determines the HTML form control used for the field. Except where noted, form field data is validated on the browser and on the server.

Simple Fields

- `forms.BooleanField()`
  - Default Widget: `CheckboxInput`
  - Resulting HTML: `<input type="checkbox" … >`
- `forms.CharField()`
  - Default Widget: `TextInput`
  - Resulting HTML: `<input type="text" … >`
- `forms.DateField()`
  - Default Widget: `DateInput`
  - Resulting HTML: `<input type="text" … >`
  - Format validation is only done on the server.
- `forms.DateTimeField()`
  - Default Widget: `DateTimeInput`
  - Resulting HTML: `<input type="text" … >`
  - Format validation is only done on the server.
- `forms.DecimalField()`
  - Default Widget: `NumberInput` (`TextInput` if `localize` is `True`)
  - Resulting HTML: `<input type="number" … >`
- `forms.DurationField()`
  - Default Widget: `TextInput`
  - Resulting HTML: `<input type="text" … >`
  - The expected input format is `DD HH:MM:SS.uuuuuu`.[52]
- `forms.EmailField()`
  - Default Widget: `EmailInput`
  - Resulting HTML: `<input type="email" … >`
- `forms.FloatField()`

52. As of Django 3.1, a comma can also be used as the separator for decimal fractions (`DD HH:MM:SS,uuuuuu`).

    - Default Widget: `NumberInput` (`TextInput` if `localize` is `True`)
    - Resulting HTML: `<input type="number" … >`
- `forms.IntegerField()`
    - Default Widget: `NumberInput` (`TextInput` if `localize` is `True`)
    - Resulting HTML: `<input type="number" … >`
- `forms.JSONField()`[53]
    - Default Widget: `Textarea`
    - Resulting HTML: `<textarea … ></textarea>`
- `forms.RegexField()`
    - Default Widget: `TextInput`
    - Resulting HTML: `<input type="text" … >`
    - Format validation is only done on the server.
- `forms.TimeField()`
    - Default Widget: `TimeInput`
    - Resulting HTML: `<input type="text" … >`
    - Format validation is only done on the server.
- `forms.URLField()`
    - Default Widget: `URLInput`
    - Resulting HTML: `<input type="url" … >`

Date and Time Field Widgets

It may seem odd that the date and time widgets use `text` instead of HTML's `date` and `time` form controls. This is because not all browsers have the same level of support for those controls.

Choice Fields

1. `forms.ChoiceField`

53. New in Django 3.1.

2. `forms.MultipleChoiceField`
3. `forms.NullBooleanField`
4. `forms.TypedChoiceField`
5. `forms.TypedMultipleChoiceField`

All of these fields except for `forms.NullBooleanField` take a `choices` argument, which must be an iterable of 2-item tuples or a function that returns such an iterable. A common practice is to create a variable holding the choices and then assign that variable to the `choices` argument. For example:

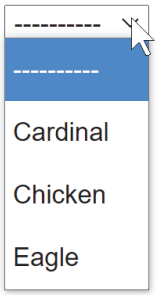
```
BIRDS = (
    (None, '----------'),
    ('cardinal', 'Cardinal'),
    ('chicken', 'Chicken'),
    ('eagle', 'Eagle')
)
```

Notice that the `BIRDS` variable includes an empty option, which you typically will include for fields that only allow the user to make one selection:

- `forms.ChoiceField(choices=BIRDS)`
    - Default Widget: `Select`
    - Resulting HTML:

```
<select name="…" id="id_…">
  <option value="" selected>----------</option>
  <option value="cardinal">Cardinal</option>
  <option value="chicken">Chicken</option>
  <option value="eagle">Eagle</option>
</select>
```

      The resulting unstyled HTML widget looks like this:



If the field is required and has a default value then you shouldn't include an empty option:

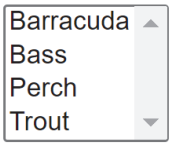
```
FISH = (
    ('barracuda', 'Barracuda'),
    ('bass', 'Bass'),
    ('perch', 'Perch'),
    ('trout', 'Trout')
)

forms.ChoiceField(choices=FISH, default='barracuda', required=True)
```

For fields that allow the user to make multiple selections, you would also typically not include an empty option:

- `forms.MultipleChoiceField(choices=FISH)`
    - Default Widget: `SelectMultiple`
    - Resulting HTML:

```
<select name="…" required id="id_…" multiple>
  <option value="barracuda">Barracuda</option>
  <option value="bass">Bass</option>
  <option value="perch">Perch</option>
  <option value="trout">Trout</option>
</select>
```

The resulting unstyled HTML widget looks like this:



The `NullBooleanField` also presents the user with choices, but it doesn't take a `choices` argument. The preset choices are "Unknown," "Yes," and "No".

- `forms.NullBooleanField()`
    - Default Widget: `NullBooleanSelect`
    - Resulting HTML:

```
<select name="…" id="id_…">
  <option value="unknown" selected>Unknown</option>
  <option value="true">Yes</option>
  <option value="false">No</option>
</select>
```

- `forms.TypedChoiceField()` – Like `ChoiceField` except the string values in the form are coerced (see page 241) into the specified type.
- `forms.TypedMultipleChoiceField()` – Like `MultipleChoiceField` except the string values in the form are coerced (see page 241) into the specified type.

File Fields

- `forms.FileField()`
    - Default Widget: `ClearableFileInput`
    - Resulting HTML (after a file is uploaded):

```
Currently:
<a href="/media/private/resumes/ndunn.pdf">resumes/ndunn.pdf</a>
<span class="clearable-file-input">
  <input type="checkbox" name="resume-clear" id="resume-clear_id">
  <label for="resume-clear_id">Clear</label>
</span><br>
Change: <input type="file" name="resume" id="id_resume">
```

The resulting unstyled HTML widget looks like this:

- `forms.ImageField()` – Like `FileField`, but the uploaded file is validated with Pillow.[54]
- `forms.FilePathField()`
    - Default Widget: `Select`
    - Outputs a `select` component listing the contents of the directory specified by the required `path` argument.

We will cover uploading files and images in the Media Files lesson (see page 369).

Additional Rarely Used Form Fields

- `forms.GenericIPAddressField()`
- `forms.SlugField()`
- `forms.UUIDField()`

54. https://pypi.org/project/Pillow/

Exercise 31: Creating a Job Application Form

45 to 60 minutes

In this exercise, you will create a job application form for a joke writer. When an applicant submits the form, Django will send an email to your email address with the data from the form.

1. Create the scaffolding:

```
(.venv) …/projects/djangojokes.com> python manage.py startapp jobs
```

2. Add the new application to the `INSTALLED_APPS` list in `djangojokes/settings.py`:

```
INSTALLED_APPS = [
    …

    # Local apps
    'common.apps.CommonConfig',
    'jobs.apps.JobsConfig',
    …
]
```

❖ E31.1. The Form

Create a new `forms.py` file within the `jobs` folder and add the following content:

Exercise Code 31.1: jobs/forms.py

```
1.   from django import forms
2.
3.   class JobApplicationForm(forms.Form):
4.       first_name = forms.CharField()
5.       last_name = forms.CharField()
6.       email = forms.EmailField()
```

Things to notice:

1. `forms` is imported from `django`.

2. `JobApplicationForm` inherits from `forms.Form`.
3. The form includes two `CharField` fields and one `EmailField` field.

❖ E31.2. The Views

You will need two views: one for the application form (a `FormView`) and one for the success page (a `TemplateView`). You are already familiar with `TemplateView` views. `FormView` (`django.views.generic.edit.FormView`) is a generic editing view for creating a form that is not based on a model.

1. Open `jobs/views.py` for editing.
2. Remove the line importing `render` and add the following lines importing `reverse_lazy` and `FormView` and `TemplateView`:

```
from django.urls import reverse_lazy
from django.views.generic import FormView, TemplateView
```

3. Import `JobApplicationForm` from the `forms.py` file you created in the same directory:

```
from .forms import JobApplicationForm
```

4. Create a new `JobAppView` that inherits from `FormView` using the following code:

```
class JobAppView(FormView):
    template_name = 'jobs/joke_writer.html'
    form_class = JobApplicationForm
    success_url = reverse_lazy('jobs:thanks')
```

   A. `template_name` identifies the template used to render the form.
   B. `form_class` identifies the `Form` class, which contains the form fields.
   C. `success_url` points to the URL to open when the form is successfully processed.

   This is all you need in the view to get the form to display and submit. The view context will contain a `form` object with the three fields specified in the `JobApplicationForm` class.

5. When the user submits the form with no errors, Django will redirect to the page identified by `success_url`. That page needs a view too. Below `JobAppView`, add this simple view:

```
class JobAppThanksView(TemplateView):
    template_name = 'jobs/thanks.html'
```

6. Your `views.py` file should now look like this:

Exercise Code 31.2: jobs/views.py

```
from django.urls import reverse_lazy
from django.views.generic import FormView, TemplateView

from .forms import JobApplicationForm

class JobAppView(FormView):
    template_name = 'jobs/joke_writer.html'
    form_class = JobApplicationForm
    success_url = reverse_lazy('jobs:thanks')

class JobAppThanksView(TemplateView):
    template_name = 'jobs/thanks.html'
```

❖ E31.3. The Templates

Create a new `jobs` folder within the `templates` folder and add these two files:[55]

55. **Don't want to type?** Copy from `starter-code/forms-and-widgets/joke_writer.html` and `starter-code/forms-and-widgets/thanks.html`.

Exercise Code 31.3: templates/jobs/joke_writer.html

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

{% block title %}Job Application: Joke Writer{% endblock %}
{% block main %}
  <h2 class="text-center">Joke Writer Application</h2>
  <div class="card border-primary m-auto mb-3 p-3" style="max-width: 40rem">
      <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button class="btn btn-success float-right">Apply</button>
      </form>
  </div>
{% endblock %}
```

Exercise Code 31.4: templates/jobs/thanks.html

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

{% block title %}Job Application Submitted{% endblock %}
{% block main %}
  <div class="card border-primary m-auto mb-3 p-3 text-center"
    style="max-width: 50rem">
    <p>Thanks! We will get back to you soon!</p>
  </div>
{% endblock %}
```

Add a link to the job application in the `pages/about_us.html` template:

Exercise Code 31.5: templates/pages/about_us.html

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

{% block title %}About Us{% endblock %}
{% block main %}
  <h2>About Us</h2>
  <p>We tell funny jokes.</p>
  <p><a href="{% url 'jobs:app' %}">Work for us.</a></p>
{% endblock %}
```

❖ E31.4. Configure URLs

Update the root `urls.py` file to hand off paths that begin with "/jobs" to the `jobs` app, and create a URLConf file for the `jobs` app with paths for `JobAppView` and `JobAppThanksView`:

Exercise Code 31.6: djangojokes/urls.py

```
     -------Lines 1 through 3 Omitted-------
4.   urlpatterns = [
5.       path('admin/doc/', include('django.contrib.admindocs.urls')),
6.       path('admin/', admin.site.urls),
7.       path('jobs/', include('jobs.urls')),
8.       path('jokes/', include('jokes.urls')),
9.       path('', include('pages.urls')),
10.  ]
```

Exercise Code 31.7: jobs/urls.py

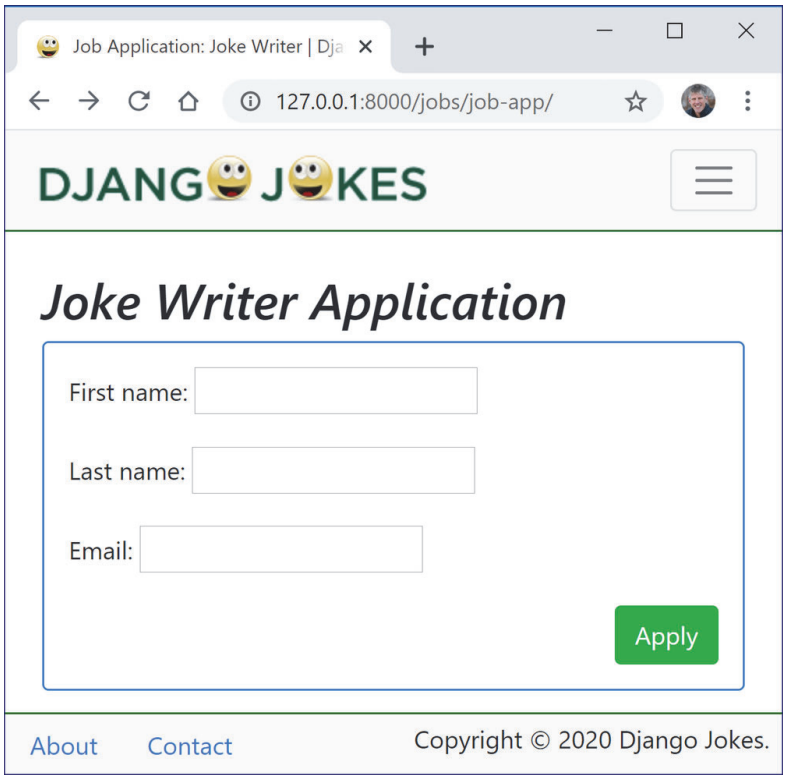
```
1.   from django.urls import path
2.
3.   from .views import JobAppView, JobAppThanksView
4.
5.   app_name = 'jobs'
6.   urlpatterns = [
7.       path('job-app/', JobAppView.as_view(), name='app'),
8.       path('job-app/thanks/', JobAppThanksView.as_view(), name='thanks'),
9.   ]
```

❖ E31.5. Try It Out

1. With `djangojokes.com` open in the terminal, run:

```
(.venv) …/projects/djangojokes.com> python manage.py runserver
```

2. In your browser, navigate to `http://127.0.0.1:8000/jobs/job-app/`. The page should look like this:



Fill out and submit the form. You should get a page with a message reading "Thanks! We will get back to you soon!".

Don't worry about the form design.

You will soon learn to make the form prettier.

At this point, there is no way you will be able to get back to applicants soon, because you haven't collected any of their information.

❖ E31.6. Emailing the Application

Add the following highlighted code to `jobs/views.py` (replace `'you@example.com'` with your email):

Exercise Code 31.8: jobs/views.py

```
1.   import html
2.   from django.urls import reverse_lazy
3.   from django.views.generic import FormView, TemplateView
4.
5.   from common.utils.email import send_email
6.   from .forms import JobApplicationForm
7.
8.   class JobAppView(FormView):
9.       template_name = 'jobs/joke_writer.html'
10.      form_class = JobApplicationForm
11.      success_url = reverse_lazy('jobs:thanks')
12.
13.      def form_valid(self, form):
14.          data = form.cleaned_data
15.          to = 'you@example.com'
16.          subject = 'Application for Joke Writer'
17.          content = f'''<p>Hey HR Manager!</p>
18.              <p>Job application received:</p>
19.              <ol>'''
20.          for key, value in data.items():
21.              label = key.replace('_', ' ').title()
22.              entry = html.escape(str(value), quote=False)
23.              content += f'<li>{label}: {entry}</li>'
24.
25.          content += '</ol>'
26.
27.          send_email(to, subject, content)
28.          return super().form_valid(form)
     -------Lines 29 through 31 Omitted-------
```

Things to notice:

1. The `send_email()` utility function you wrote is imported from `common.utils.email`.
2. The `form_valid()` function is executed when the form passes validation. `form.cleaned_data` will contain all the data submitted in the form. Depending on how the form fields are implemented, some of the data may have been "cleaned," which is why the variable is called `cleaned_data`. We had you immediately assign `cleaned_data` to `data`, both to have a simpler variable name and to make it clear that you have access to this data.
3. You set values for `to`, `subject`, and `content` and pass those variables to `send_email()`. To create the `content` variable, you loop through all the items in the cleaned data, outputting key-value pairs in an HTML list.

- Notice that you use `html.escape` to escape the values entered by the user. That's to protect against malicious HTML.

4. Finally, you return `super().form_valid(form)`, which will take care of redirecting the page to the "jobs:thanks" named URL pattern (as per `success_url`).

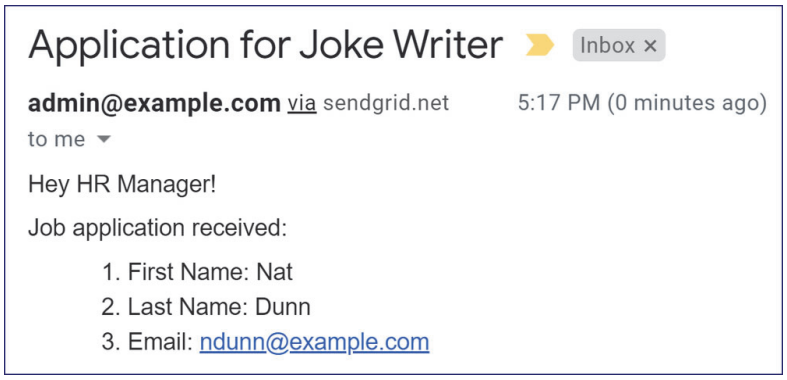
❖ E31.7. Try It Out Again

python manage.py runserver

From here on out, when we say "start up the server" or "run the server," you should run:

```
python manage.py runserver
```

1. With `djangojokes.com` open in the terminal, start up the server.
2. In your browser, navigate to `http://127.0.0.1:8000/jobs/job-app/`, fill out and submit the form. Then, check your email. You should have received something like this:



Git Commit

Commit your code to Git.