Module-3: Django Admin Interfaces and Model Forms

Activating Admin Interfaces **(P83-85)** - Using Admin Interfaces **(P85-91)** - Customizing Admin Interfaces **(P91-93)** - Reasons to use Admin Interfaces **(P94)**.

Form Processing **(P95-98)** - Creating Feedback forms **(P98-101)** - Form submissions **(P101-102)**, custom validation **(P103-105)**, creating Model Forms **(P105-106)** - URLConf Tricks **(P107-120)**, Other URLConfs **(P120-122)**.

Textbook 1: Chapters 6, 7 and 8

Activating Admin Interfaces (P83-85)

- For a certain class of Web sites, an admin interface is an essential part of the infrastructure.
- This is a Web-based interface, limited to trusted site administrators, that **enables the addition, editing, and deletion of site content**.
- The interface you use to **post to your blog, the back-end site managers use to moderate reader-generated comments, the tool your clients use to update the press releases on the Web site you built for them**—these are all **examples** of admin interfaces.
- There are **three steps** you'll need to follow to activate it:
  1. **Add admin metadata to our models.** Not all models can (or should) be editable by admin users, so we need to "mark" models that should have an admin interface.

     ```
     class Book(models.Model):
     title = models.CharField(maxlength=100)
     authors = models.ManyToManyField(Author)
     publisher = models.ForeignKey(Publisher)
     publication_date = models.DateField()
     num_pages = models.IntegerField(blank=True, null=True)

     class Admin: pass
     ```

  2. Install the admin application. Do this by adding ***django.contrib.admin*** to your ***INSTALLED_APPS*** setting and running ***python manage.py syncdb***. This second step will install the extra database tables the admin interface uses.
  3. Add the URL pattern to your ***urls.py***. If you're still using the one created by ***startproject***, the admin URL pattern should be already there, but commented out. Either way, your URL patterns should look like the following:

     ```
     from django.conf.urls.defaults import *
     urlpatterns = patterns('', (r'^admin/', include('django.contrib.admin.urls')),
     )
     ```

     Now run ***python manage.py runserver*** to start the development server – will see like this:

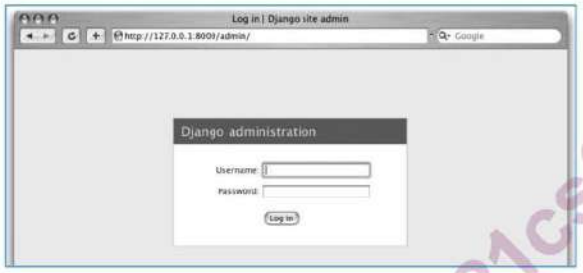
     ```
     Validating models...
     0 errors found.

     Django version 0.96, using settings 'mysite.settings'
     Development server is running at http://127.0.0.1:8000/
     Quit the server with CONTROL-C.
     ```

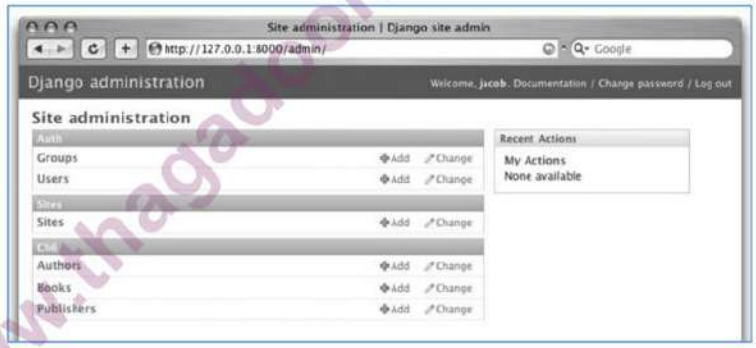
- Now you can visit the URL given to you by Django (**http://127.0.0.1:8000/admin/** in the preceding example), log in, and play around.

Using Admin Interfaces (P85-91)

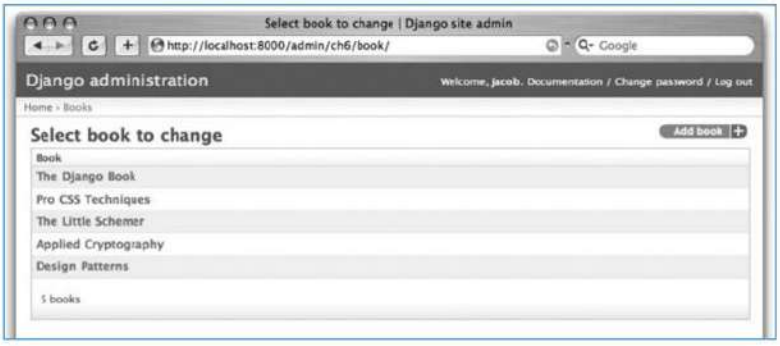
- The admin interface is designed to be used by nontechnical users, and as such it should be pretty self-explanatory. Nevertheless, a few notes about the features of the admin interface are in order. The first - login screen, as shown in Figure.



- Use the ***username*** and ***password*** we set up when we first added our ***superuser*** account.
- Once logged in, see that we can manage
  - Users
  - groups, and
  - permissions
- Each object given an Admin declaration shows up on the main index page, as shown in Figure:

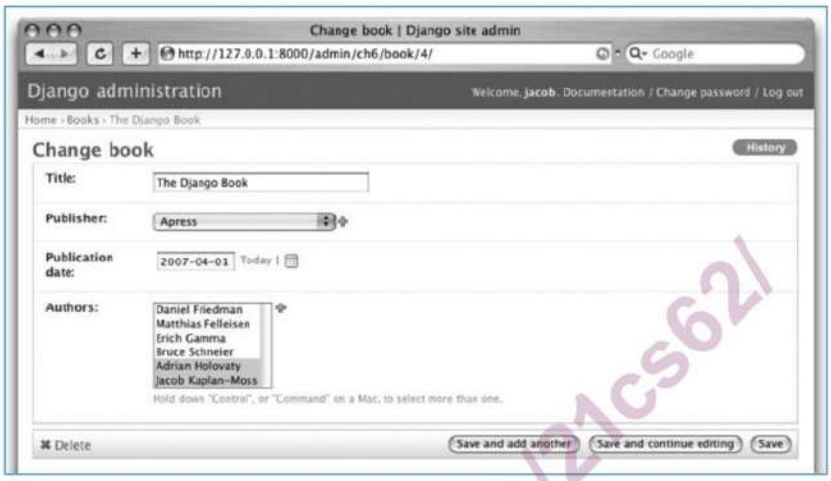


- Links to add and change objects lead to two pages we refer to as object ***change lists*** and ***edit forms***. **Change lists are essentially index pages of objects** in the system, as shown in Figure:

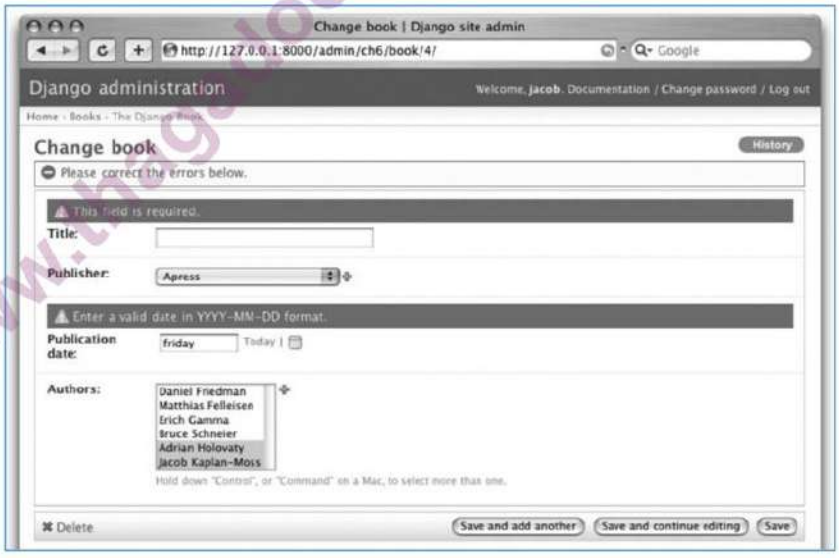


- A number of **options control** which fields appear on these lists and the appearance of extra features like **date drill-downs, search fields, and filter interfaces**.

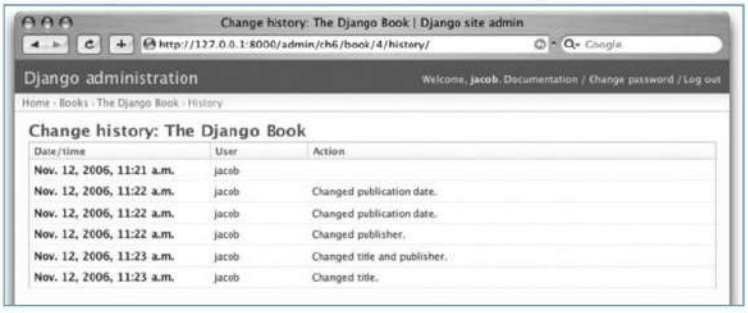
- **Edit forms** are **used to modify existing objects and create new ones** shown in Figure:



- **Each field defined in model appears** here, and **notice that fields of different types get different widgets** (e.g., date/time fields have calendar controls, foreign keys use a select box, etc.).
- The **admin interface also handles input validation.**
- Try leaving a required field blank or putting an invalid time into a time field – generate errors when try to save, as shown in Figure:



- **When edit an existing object**, a **History button will show in the upper-right corner** of the window. **Every change made through the admin interface is logged, and you can examine this log by clicking the History button ---------→**



- When **delete an existing object**, the **admin interface asks to confirm the delete action to avoid costly mistakes**. Deletions also cascade; the **deletion confirmation page shows all the related objects that will be deleted as well** as shown in Figure:

Users, Groups, and Permissions

- Since ***logged in as a superuser***, have ***access to create, edit, and delete any object***.
- The ***admin interface has a user permissions system*** that can use to give other users access only to the portions of the interface that they need.
- ***Edit these users and permissions through the admin interface*** like any other object.
- The ***link to the User and Group models*** is there on the **admin index** along with all the objects defined ourselves.
- **User objects** have the **standard username, password, e-mail,** and **real name fields** we might expect, along with a set of fields that define what the user is allowed to do in the admin interface.
- There's a set of **three flags**:
  - The **"is active"** flag controls whether the user is active at all. If this **flag is off**, the ***user has no access to any URLs that require login***.
  - The **"is staff"** flag controls **whether the user is allowed to log in to the admin interface** (i.e., whether that user is considered a "staff member" in the organization). Since this **same user system can be used to control access to public** (i.e., **nonadmin**) sites, **this flag differentiates between public users and administrators**.
  - The **"is superuser"** flag gives the **user full, unfettered access to every item** in the **admin interface**; regular permissions are ignored.
- "Normal" admin users—that is, active, nonsuperuser staff members—are granted access that depends on a set of assigned permissions.
- **Each object editable through the admin interface has three permissions**:
  - Create permission
  - Edit permission, and
  - Delete permission.
- Assigning permissions to a user grants the user access to do what described by those permissions.
- We can **also assign users to groups**.
- A **group** is **a set of permissions to apply to all members of that group. Groups** are **useful for granting identical permissions to a large number of users.**

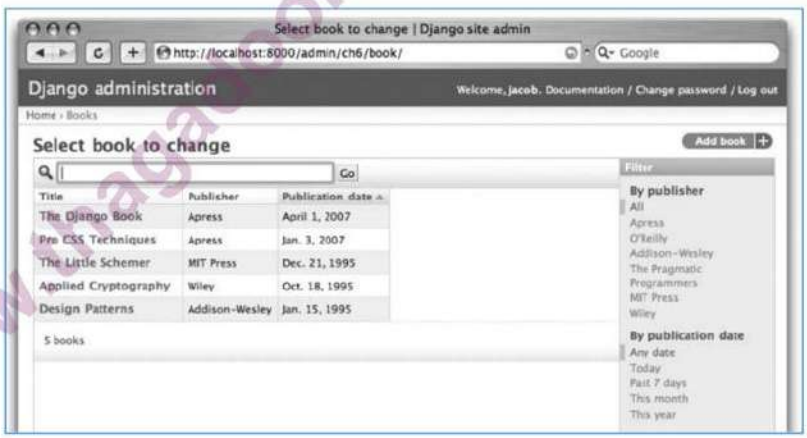
Customizing Admin Interfaces

- Customize the way the admin interface looks and behaves in a number of ways.
- The **change list for our books shows only the string representation of the model** we added to its **__str__**.
- This **works fine for just a few books**, but if **we had hundreds or thousands of books**, it would be **very hard to locate a single needle in the haystack.**
- Can easily add some **display**, **searching**, and **filtering** functions to this interface.
- Change the **Admin** declaration as follows:

```
class Book(models.Model):
    title = models.CharField(maxlength=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    class Admin:
        list_display = ('title', 'publisher', 'publication_date')
        list_filter = ('publisher', 'publication_date')
        ordering = ('-publication_date',)
        search_fields = ('title',)
```

- These four lines of code dramatically change our list interface, as shown in Figure:



- Each of those lines instructed the admin interface to construct a different piece of this interface:
  - The ***list_display*** option **controls which columns appear in the change list table**. By default, the change list displays only a single column that contains the object's string representation. Here, we've changed that to show the title, publisher, and publication date.
  - The ***list_filter*** option **creates the filtering bar on the right side of the list**. We've allowed filtering by date (which allows **to see only books published in the last week, month**, etc.) and by publisher. The filters show up as long as there are at least two values to choose from. You **can instruct the admin interface to filter by any field**, **but foreign keys, dates, Booleans, and fields with choices attribute work best.**
  - The ***ordering*** option controls the order in which the objects are presented in the admin interface (In this example, ordering by **publication date**, with the most recent first.)

  - Finally, the ***search_fields*** option creates a **field that allows text searches**. It allows **searches by the title field** (so you could type Django to show all books with "Django" in the title).
- Using these options, with only **a few lines of code, make a very powerful, production-ready interface for data editing.**

Customizing the Admin Interface's Look and Feel

- It's easy to change, though, using Django's template system. The Django admin site is powered by Django itself, and its interfaces use Django's own template system.
- The **TEMPLATE_DIRS** setting **specifies a list of directories to check when loading Django templates.** To **customize Django's admin templates**, simply **copy the relevant stock admin template** from the Django distribution into one of the directories pointed to by **TEMPLATE_DIRS**.
- The **admin site finds** the **"Django administration" header** by looking for the template **admin/base_site.html**.
- By default, this template lives in the **Django admin template directory**, **django/contrib/admin/templates**, which can find by looking in Python site-packages directory, or wherever Django was installed.
- **To customize** this **base_site.html** template, **copy that template into an admin subdirectory** of **whichever directory** we're using in **TEMPLATE_DIRS**.
- For example, if **TEMPLATE_DIRS** includes **"/home/mytemplates"**, then **copy django/contrib/admin/templates/admin/base_site.html → /home/mytemplates/admin/base_site.html.**
- Next, edit the new **admin/base_site.html** file to replace the generic Django text with own site's name as to see fit.
- **Any of Django's default admin templates can be overridden. To override a template**, copy **base_site.html** from the default directory into custom directory and make changes to the copy.

Customizing the Admin Index Page

- By default, **Django admin index** displays all available applications, according to **INSTALLED_APPS** setting, sorted by the name of the application. If we want to change this order to **make it easier to find the applications looking for**. The **index is probably the most important page of the admin interface**, so it should be easy to use.
- The template to customize is **admin/index.html**. (Remember to copy admin/index.html to your custom template directory as in the previous example.)
- **Edit the file, and see** it uses a template tag called **{% get_admin_app_list as app_list %}. This tag retrieves every installed Django application**. Instead of using the tag, you can hard-code links to object-specific admin pages in whatever way you think is best.
- Django offers another shortcut - Run the command "**$ python manage.py admininde**x"to get a chunk of template code for inclusion in the admin index template. It is a useful starting point.

Reasons to use Admin Interfaces (When and Why to Use the Admin Interface) (P-94)

- We think Django's admin interface is outstanding - call it one of Django's "killer features."
- The **admin interface is extremely useful for editing data**. If **have any sort of data entry tasks**, the admin interface simply can't be beat.
- Django's admin interface especially shines when nontechnical users need to be able to enter data; that's the purpose behind the feature.
- At the newspaper where Django was first developed, development of a typical online feature—a special report on water quality in the municipal supply, say—goes something like this:
    - The reporter responsible for the story meets with one of the developers and goes over the available data.
    - The developer designs a model around this data and then opens up the admin interface to the reporter.
    - While the reporter enters data into Django, the programmer can focus on developing the publicly accessible interface.
- The admin interface useful in a few other cases:
    - **Inspecting data models**: The first thing we do when we've defined a new model is to call it up in the admin interface and enter some dummy data. This is usually when we find any data modeling mistakes; having a graphical interface to a model quickly reveals problems.
    - **Managing acquired data**: There's little actual data entry associated with a site like http://chicagocrime.org, **since most of the data comes from an automated source**. However, when problems with the automatically acquired data crop up, it's useful to be able to go in and edit that data easily.

Form Processing **(P95-98)** - Creating Feedback forms **(P98-101)** - Form submissions **(P101-102)**, custom validation **(P103-105)**, creating Model Forms **(P105-106)** - URLConf Tricks **(P107-120)**, Other URLConfs **(P120-122)**.

Form Processing (P95-98)

- Next piece of the puzzle: **building views that take input from readers**. Start by making a simple **search form "by hand"** and looking at **how to handle data submitted from the browser**. From there, the control move on to using Django's forms framework.
- The **Web is all about search**. Two of the Net's **biggest success stories, Google and Yahoo**, built their multibillion-dollar businesses around search. Nearly every site sees a large percentage of traffic coming to and from its search pages. Often **the difference between a site's success or failure is the quality of its search**.
- Start by adding the search view to our **URLconf (mysite.urls)** - adding something like ***(r'^search/$', 'mysite.books.views.search')*** to the set of URL patterns.
- Next, we'll write this search view into our view module ***(mysite.books.views):***

```
from django.db.models import Q
from django.shortcuts import render_to_response
from mysite.books.models import Book

def search(request):
    query = request.GET.get('q', '')
    if query:
        qset = (
            Q(title__icontains=query) |
            Q(authors__first_name__icontains=query) |
            Q(authors__last_name__icontains=query)
        )
        results = Book.objects.filter(qset).distinct()
    else:
        results = []
    return render_to_response("books/search.html", {
        "results": results,
        "query": query
    })
```

- First, there's **request.GET.** This is how you access GET data from Django; POST data is accessed through a similar **request.POST** object. These **objects behave exactly like standard Python dictionaries**.
- GET and POST are the two methods that browsers use to send data to a server. Most of the time, you'll see them in HTML form tags: ***<form action="/books/search/" method="get">***
- This instructs the browser to submit the form data to the URL /books/search/ using the GET method.
- So the line ***query = request.GET.get('q', '')*** looks for a GET parameter named q and returns an empty string if that parameter wasn't submitted.
- The ***get()*** method here is the one that every Python dictionary has - it is **not safe to assume** that ***request.GET*** contains a ***'q'*** key, so we use get***('q', '')*** **to provide a default fallback value of**

'' (the empty string). If we merely accessed the variable using ***request.GET['q'],*** **that code would raise a KeyError** if ***q*** wasn't available in the GET data.

- ***Q (QuerySet)*** objects are used to build up complex queries—in this case, searching for any books where ***either the title*** or ***the name of one of the authors*** matches the search query.
- In these queries, ***icontains*** is **a case-insensitive search** that uses the **SQL LIKE** operator in the underlying database.
- Since ***searching against a many-to-many field***, it's possible for the ***same book to be returned more than once by the query*** (e.g., a book with two authors who both match the search query). Adding ***.distinct()*** to the ***filter lookup eliminates any duplicate results***.
- There's still no template for this search view, however. This should do the trick:

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.01//EN">
<html lang="en">
<head>
  <title>Search {% if query %}Results{% endif %}</title>
</head>
<body>
  <h1>Search</h1>
  <form action="." method="GET">
    <label for="q">Search: </label>
    <input type="text" name="q" value="{{ query|escape }}">
    <input type="submit" value="Search">
  </form>

  {% if query %}
    <h2>Results for "{{ query|escape }}":</h2>

    {% if results %}
      <ul>
      {% for book in results %}
        <li>{{ book }}</li>
      {% endfor %}
      </ul>
    {% else %}
      <p>No books found</p>
    {% endif %}
  {% endif %}
</body>
</html>
```

The "Perfect Form"

- Forms can often be a major cause of frustration for the users of your site. Let's consider the behavior of a hypothetical perfect form:
  - Forms should ask the user for some information, accessibility and usability matter here, so **smart use of the HTML element and useful contextual help are important.**
  - The submitted data should be subjected to extensive validation. The golden rule of Web application security is "never trust incoming data," so **validation is essential**.

- If the user has made any mistakes, the form should be **redisplayed with detailed, informative error messages.** The original data should be prefilled, to save the user from having to re-enter everything.
- The form should continue to redisplay until all of the fields have been correctly filled.

- Constructing the perfect form seems like a lot of work! Thankfully, **Django's forms framework is designed to do most of the work for you**. You provide a description of the form's fields, the validation rules, and a simple template, and Django does the rest. **The result is a "perfect form" with very little effort**.

Creating Feedback forms (P98-101)

- The **best way to build a site that people love is to listen to their feedback**. Many sites appear to have forgotten this; they hide their contact details behind layers of FAQs, and they seem to make it as difficult as possible to get in touch with an actual human being.
- When a site has millions of users, this may be a reasonable strategy. When we are trying to build up an audience, though, we should actively encourage feedback at every opportunity.
- Let's **build a simple feedback form and use it to illustrate Django's forms framework in action**.
- Forms in Django are created in a similar way to models: declaratively, using a Python class.
- Here's the class for our simple form. By convention, we'll insert it into a new forms.py file within our application directory.

```
from django import newforms as forms

TOPIC_CHOICES = (
    ('general', 'General enquiry'),
    ('bug', 'Bug report'),
    ('suggestion', 'Suggestion'),
)

class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField()
    sender = forms.EmailField(required=False)
```


- A Django form is a **subclass** of ***django.newforms.Form***, just as a Django model is a subclass of ***django.db.models.Model***. The ***django.newforms*** module also contains a number of Field classes (a full list is available in Django's documentation at http://www.djangoproject.com/documentation/0.96/newforms/).
- Our **ContactForm** consists of **three fields**:
  - a **topic**, which is a choice among three options;
  - a **message**, which is a character field; and
  - a **sender**, which is an email field and is optional (because even anonymous feedback can be useful).
- There are a number of other field types available, and you can write your own if they don't cover your needs.
- The **form object itself knows how to do a number of useful things**. It **can validate a collection of data**, it can **generate its own HTML "widgets,"** it can construct a set of useful error messages and it can even draw the entire form for us.

- In ***views.py***:

```
from django.shortcuts import render_to_response
from mysite.books.forms import ContactForm

def contact(request):
    form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

and in contact.html:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html lang="en">
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>
    <form action="." method="POST">
        <table>
            {{ form.as_table }}
        </table>
        <p><input type="submit" value="Submit"></p>
    </form>
</body>
</html>
```

- {{ **form.as_table** }} --→ form is our **ContactForm** instance, as passed to ***render_to_response. as_table*** is a method on that object that renders the form as a sequence of table rows (as_"**ul**" or "**p**" can also be used).
- The generated HTML looks like this:

```
<tr>
    <th><label for="id_topic">Topic:</label></th>
    <td>
        <select name="topic" id="id_topic">
            <option value="general">General enquiry</option>
            <option value="bug">Bug report</option>
            <option value="suggestion">Suggestion</option>
        </select>
    </td>
</tr>
<tr>
    <th><label for="id_message">Message:</label></th>
    <td><input type="text" name="message" id="id_message" /></td>
</tr>
<tr>
    <th><label for="id_sender">Sender:</label></th>
    <td><input type="text" name="sender" id="id_sender" /></td>
</tr>
```

- Note that the <table> and <form> tags are not included; you need to define those yourself in the template, which gives you control over how the form behaves when it is submitted.

- Label elements are included, making forms accessible out of the box.
- Our form is currently using a **<input type="text">** widget for the message field. We don't want to restrict our users to a single line of text, so we'll swap in a **<textarea>** widget instead:

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)
```

- The forms framework separates out the presentation logic for each field into a set of widgets.
- **Each field type has a default widget**, but you **can easily override the default**, or **provide a custom widget of your own.**
- Now, **submitting the form does not actually do anything.** Let's hook in our validation rules:

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

- A form instance can be in one of **two states**: **bound** or **unbound**.
  - A **bound** instance is **attached to a dictionary** (or dictionary-like object) and **knows how to validate and redisplay the data from it.**
  - An **unbound** form has **no data associated with it** and **simply knows how to display itself.**
- Try clicking **Submit** on the **blank form**. The **page should redisplay, showing a validation error that informs us that our message field is required.**
- **Try entering an invalid email address as well.** The **EmailField knows how to validate email addresses**, at least to a reasonable level of doubt.

Form submissions - Processing the Submission (P101-102)

- Once the user has filled the form to the point that it passes our validation rules, we need to do something useful with the data.
- In this case, we want to construct and send an email containing the user's feedback. We'll use **Django's email package** to do this.
- First, though, we need to tell if the data is valid, and if it is, we need access to the validated data. The **forms framework does more than just validate the data; it also converts it into Python types.**
- Our **contact form only deals with strings**, but if we were to use an **IntegerField** or **DateTimeField**, the forms framework would ensure that we got back a **Python integer** or **datetime object**, respectively.
- To tell whether a form is bound to valid data, call the **is_valid()** method:

```
form = ContactForm(request.POST)
if form.is_valid():
    # Process form data
```

- Now we need access to the data. We could pull it straight out of **request.POST** - miss the type conversions performed by the forms framework. Instead, we use **form.clean_data:**

```
if form.is_valid():
    topic = form.clean_data['topic']
    message = form.clean_data['message']
    sender = form.clean_data['sender']
    # ...
```

- Finally, we need to record the user's feedback. The easiest way to do this is to **email it to a site administrator**. We can do that using the **send_mail** function:

```
from django.core.mail import send_mail

# ...

send_mail(
    'Feedback from your site, topic: %s' % topic,
    message, sender,
    ['administrator@example.com']
)
```

- The **send_mail** function has **four** required arguments: **the email subject**, the **email body**, the **"from" address**, and **a list of recipient addresses**. **send_mail** is a convenient wrapper around Django's **EmailMessage** class, **which provides advanced features such as attachments, multipart emails, and full control over email headers.** Having sent the feedback email, redirect user to a static confirmation page. **The finished view function looks like this:**

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from django.core.mail import send_mail
from forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            topic = form.cleaned_data['topic']
            message = form.cleaned_data['message']
            sender = form.cleaned_data.get('sender', 'noreply@example.com')
            send_mail(
                'Feedback from your site, topic: %s' % topic,
                message, sender,
                ['administrator@example.com']
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render_to_response('contact.html', {'form': form})
```

Custom validation (P103-105)

- There are a number of ways to hook custom validation into a Django form.
- If our rule is something we will reuse repeatedly, we can create a custom field type.
- Most **custom validations are one-off affairs**, though, and **can be tied directly to the form class**.
- We want additional validation on the message field, so we need to add a **clean_message** method to our form:

```
class ContactForm(forms.Form):
    topic = forms.ChoiceField(choices=TOPIC_CHOICES)
    message = forms.CharField(widget=forms.Textarea())
    sender = forms.EmailField(required=False)

    def clean_message(self):
        message = self.clean_data.get('message', '')
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

- This new method will be called after the default field validator (in this case, the validator for a required **CharField**). Because the field data has already been partially processed, we need to pull it out of the form's **clean_data** dictionary.
- We simply use a combination of **len()** and **split()** to **count the number of words**. If the user has **entered too few words**, we raise a **ValidationError**.
- The string attached to this exception will be displayed to the user as an item in the error list.
- It is **important that we explicitly return the value for the field at the end of the method**. This allows us to modify the value (or convert it to a different Python type) within our custom validation method. **If we forget, the return statement** and then **None** will be returned, and the **original value will be lost**.

A Custom Look and Feel (P105-106)

- The quickest way to customize the **form's presentation is with CSS**. The list of errors in particular could do with some visual enhancement, and the **<ul>** has a class attribute of **errorlist** for that exact purpose. The following CSS really makes our errors stand out:

```
<style type="text/css">
    ul.errorlist {
        margin: 0;
        padding: 0;
    }
    .errorlist li {
        background-color: red;
        color: white;
        display: block;
        font-size: 10px;
        margin: 0 0 3px;
        padding: 4px 5px;
    }
</style>
```

- While it's convenient to have our form's HTML generated for us, in many cases the default rendering won't be right for our application.
- **{{ form.as_table }}** and friends are useful shortcuts while we develop our application, but everything about the way a form is displayed can be overridden, mostly within the template itself.
- Each field widget **(<input type="text">, <select>, <textarea>**, or similar) can be rendered individually by accessing **{{ form.fieldname }}**.

- Any errors associated with a field are available as **{{ form.fieldname.errors }}**. We can use these form variables to construct a custom template for our contact form:

```
<form action="." method="POST">
    <div class="fieldWrapper">
        {{ form.topic.errors }}
        <label for="id_topic">Kind of feedback:</label>
        {{ form.topic }}
    </div>
    <div class="fieldWrapper">
        {{ form.message.errors }}
        <label for="id_message">Your message:</label>
        {{ form.message }}
    </div>
    <div class="fieldWrapper">
        {{ form.sender.errors }}
        <label for="id_sender">Your email (optional):</label>
        {{ form.sender }}
    </div>
    <p><input type="submit" value="Submit"></p>
</form>
```

- **{{ form.message.errors }}** will display as a **<ul class="errorlist">** if errors are present and a blank string if the field is valid (or the form is unbound).
- We can also treat **form.message.errors** as a **Boolean** or **even iterate over** it as a list, for example

```
<div class="fieldWrapper{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
        <ol>
        {% for error in form.message.errors %}
            <li><strong>{{ error|escape }}</strong></li>
        {% endfor %}
        </ol>
    {% endif %}
    {{ form.message }}
</div>
```

- In the case of validation errors, this will add an "errors" class to the containing and display the **list of errors in an ordered list**.

Creating Model Forms (P105-106)

- An important principle in software development that Django tries to adhere to is **Don't Repeat Yourself (DRY).**
- Our **Publisher** model class says that a publisher has a **name**, **address**, **city**, **state_province**, **country**, and **website**.
- Duplicating this information in a form definition would break the DRY rule. Instead, we can use a useful shortcut: **form_for_model():**

```
from models import Publisher
from django.newforms import form_for_model

PublisherForm = form_for_model(Publisher)
```

PublisherForm is a Form subclass, just like the ContactForm class we created manually earlier on. We can use it in much the same way:

```
def add_publisher(request):
    if request.method == 'POST':
        form = PublisherForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/add_publisher/thanks/')
    else:
        form = PublisherForm()
    return render_to_response('add_publisher.html', {'form': form})
```

- The **add_publisher.html** file is almost identical to our original **contact.html** template, so it has been omitted.
- Since forms derived from models are often used to save new instances of the model to the database, the form class created by **form_for_model** includes a convenient **save()** method.
- **form_for_instance()** is a related method that can create a pre-initialized form from an instance of a model class. This is useful for creating "edit" forms.

==========