**SKILL ADVANCED COURSE – 2**

**SALESFORCE**

Submitted by

21A51A05D1 - P.V.S.K. Mourya

21A51A05D2 - CH. Himanth Kumar

21A51A05D3 - T. Pujitha

22A55A0505 - B. Ganesh

22A55A0506 - CH. Tanusri

22A55A0507 - G. Bhavani Prasad



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ADITYA INSTITUTE OF TECHNOLOGY AND MANAGEMENT

(Approved by AICTE, New Delhi, affiliated to JNTUK. Accredited by NBA & NAAC)

(AUTONOMUS, K.KOTTURU, TEKKALI – 532201

ADITYA INSTITUTE OF TECHOLOGY AND MANAGEMENT

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



CERTIFICATE

This is to certify that the Skill Oriented Course – 1 entitled “SALESFORCE” is being submitted by P.V.S.K. Mourya, CH. Himanth kumar, T. Pujitha , B. Ganesh, CH. Tanusri, G. Bhavani Prasad in partial fulfilment of requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering, Aditya Institute of Technology and Management, Tekkali is a record of bonafide work carried out by them under my guidance and supervision during the academic year 2022 – 2023.

**Signature of the Co-ordinator**  **Signature of Head of the Department**

Sri. G. Surya. Pavan Kumar, M. Tech Dr. Y. Ramesh M. Tech. Ph.D.

Asst. Professor Head of the Department

Department of CSE Department of CSE

**List of contents:**

* **Salesforce Lightning Experience**
  + **Overview of Lightning Experience**
  + **Migrating from Classic to Lightning**
  + **Customizing the Lightning Interface**
  + **Building Lightning Components**
  + **Leveraging Lightning for Enhanced Productivity**
* **Super Badge**
* **Process Automation Specialist**

**SalesForce Lightning Experience**

**INTRODUCTION**

Creating forms in Django, is really similar to creating a model. Here again, we just need to inherit from Django class and the class attributes will be the form fields.

Let's add a **forms.py** file in myapp folder to contain our app forms. We will create a login form.

**myapp/forms.py**

#-\*- coding: utf-8 -\*-

from django import forms

class LoginForm(forms.Form):

user = forms.CharField(max\_length = 100)

password = forms.CharField(widget = forms.PasswordInput())

As seen above, the field type can take "widget" argument for html rendering; in our case, we want the password to be hidden, not displayed. Many others widget are present in Django: **DateInput** for dates, **CheckboxInput** for checkboxes, etc.

# Search

A common task for web applications is to search some data in the database with user input. In a simple case, this could be filtering a list of objects by a category. A more complex use case might require searching with weighting, categorization, highlighting, multiple languages, and so on. This document explains some of the possible use cases and the tools you can use. We’ll refer to the same models used in [Making queries](https://docs.djangoproject.com/en/4.0/topics/db/queries/).

## **Use Cases**

### Standard textual queries

### Text-based fields have a selection of matching operations. For example, you may wish to allow lookup up an author like so:

**>>>** Author.objects.filter(name\_\_contains='Terry')

[<Author: Terry Gilliam>, <Author: Terry Jones>]

This is a very fragile solution as it requires the user to know an exact substring of the author’s name. A better approach could be a case-insensitive match ([icontains](https://docs.djangoproject.com/en/4.0/ref/models/querysets/" \l "std-fieldlookup-icontains)), but this is only marginally better.

### A database’s more advanced comparison functions[¶](https://docs.djangoproject.com/en/4.0/topics/db/search/#a-database-s-more-advanced-comparison-functions)

If you’re using PostgreSQL, Django provides [a selection of database specific tools](https://docs.djangoproject.com/en/4.0/ref/contrib/postgres/search/) to allow you to leverage more complex querying options. Other databases have different selections of tools, possibly

via plugins or user-defined functions. Django doesn’t include any support for them at this time. We’ll use some examples from PostgreSQL to demonstrate the kind of functionality databases may have.

**Searching in other databases**

All of the searching tools provided by [django.contrib.postgres](https://docs.djangoproject.com/en/4.0/ref/contrib/postgres/" \l "module-django.contrib.postgres" \o "django.contrib.postgres: PostgreSQL-specific fields and features) are constructed entirely on public APIs such as [custom lookups](https://docs.djangoproject.com/en/4.0/ref/models/lookups/) and [database functions](https://docs.djangoproject.com/en/4.0/ref/models/database-functions/). Depending on your database, you should be able to construct queries to allow similar APIs. If there are specific things which cannot be achieved this way, please open a ticket.

In the above example, we determined that a case insensitive lookup would be more useful. When dealing with non-English names, a further improvement is to use [unaccented comparison](https://docs.djangoproject.com/en/4.0/ref/contrib/postgres/lookups/#std-fieldlookup-unaccent):

**>>>** Author.objects.filter(name\_\_unaccent\_\_icontains='Helen')

[<Author: Helen Mirren>, <Author: Helena Bonham Carter>, <Author: Hélène Joy>]

This shows another issue, where we are matching against a different spelling of the name. In this case we have an asymmetry though - a search for Helen will pick up Helena or Hélène, but not the reverse. Another option would be to use a [trigram\_similar](https://docs.djangoproject.com/en/4.0/ref/contrib/postgres/lookups/" \l "std-fieldlookup-trigram_similar) comparison, which compares sequences of letters.

For example:

**>>>**Author.objects.filter(name\_\_unaccent\_\_lower\_\_trigram\_similar='Hélène')

[<Author: Helen Mirren>, <Author: Hélène Joy>]

Now we have a different problem - the longer name of “Helena Bonham Carter” doesn’t show up as it is much longer. Trigram searches consider all combinations of three letters, and compares how many appear in both search and source strings. For the longer name, there are more combinations that don’t appear in the source string, so it is no longer considered a close match.

The correct choice of comparison functions here depends on your particular data set, for example the language(s) used and the type of text being searched. All of the examples we’ve seen are on short strings where the user is likely to enter something close (by varying definitions) to the source data.

### **Document-based search**

Standard database operations stop being a useful approach when you start considering large blocks of text. Whereas the examples above can be thought of as operations on a string of characters, full text search looks at the actual words. Depending on the system used, it’s likely to use some of the following ideas:

* Ignoring “stop words” such as “a”, “the”, “and”.
* Stemming words, so that “pony” and “ponies” are considered similar.
* Weighting words based on different criteria such as how frequently they appear in the text, or the importance of the fields, such as the title or keywords, that they appear in.

There are many alternatives for using searching software, some of the most prominent are [Elastic](https://www.elastic.co/) and [Solr](https://solr.apache.org/). These are full document-based search solutions. To use them with data from Django models, you’ll need a layer which translates your data into a textual document, including back-references to the database ids. When a search using the engine returns a certain document, you can then look it up in the database. There are a variety of third-party libraries which are designed to help with this process.

**Getting Date From the Request Object**

we introduced Http Request objects in chapter 3 when we first covered view functions, but we didn’t have much to say about them at the time. Recall that each view function takes an Http Request object as its parameter, as in our hello() view: Django.http import Http response def hello (request): return, have a number of interesting attributes and methods that you should familiarize yourself with, so that you know what’s possible. You can use these attributes to get information about the current request (i.e., the user/web browser that’s loading the current page on your Django – powered site), at the time the view function is executed.

**Information About the URL**

Http Request objects contain several pieces of information about currently.

|  |  |  |
| --- | --- | --- |
| **Attribute/method** | **Description** | **Example** |
| Request.path | The full path,not including the domain but including the leading slash. | “/hello/” |
| Request.get\_host () | The host (i.e., the “domain,” in common paralance). | “127.0.0.1:8000” or “www.example.com” |
| Request.get\_full\_path() | The path, plus a query string(if available). | “/hello/”?print=true” |
| Request.is\_secure() | True if the request was made via HTTPS. Otherwise, False | True or False |

# BAD!

def current\_url\_view\_bad(request):

return HttpResponse("Welcome to the page at /current/")

# GOOD def current\_url\_view\_good(request):

return HttpResponse("Welcome to the page at %s" % request.path)

**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:

* It should ask the user for some information, obviously. Accessibility and usability matter here, so smart use of the HTML <label> 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 reenter 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, validation rules, and a simple template, and Django does the rest. The result is a “perfect form” with very little effort.

Creating a Feedback Form – 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 your site has millions of users, this may be a reasonable strategy. When you’re trying to build up an audience, though, you 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.

We’ll start by adding adding (r’^contact/$’, ‘mysite.books.views.contact’) to the URLconf, then defining our form. 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’ ‘report’),

(‘suggestion’, ‘Suggestion’)

)

Class Contact Form (forms. Form):

topic = forms. Choice Field(choices = TOPIC\_ CHOICES)

message = forms. Char Field()

sender = forms. Email Field (required = false)

“New” Forms? What? – When Django was first released to the public, it had a complicated, confusing forms system. It made producing forms far too difficult, so it was completely rewritten and is now called “newforms.” However, there’s still a fair amount of code that depends on the “old” form system, so for the time being Django ships with two form packages.

As we write this book, Django’s old form system is still available as django.forms and the new form package as django.newforms. At some point that will change and django.forms will point to the new form package. However, to make sure the examples in this book work as widely as possible, all the examples will refer to django.newforms.

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 document: <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, if we’re feeling lazy, it can even draw the entire form for us. Let’s hook it into a view and see it in action. In views.py:

from django. db. models import Q

from django. shortcuts import render\_to\_response

from models import Book

from forms import Contact Form

Def search (request):

Query = request. GET.get(‘q’,”)

If query:

Q set = (

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

})

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>

The most interesting line here is {{ 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 and as\_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. At the moment, submitting the form doesn’t 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 constructed with 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.

**Setting Initial Data** – Passing data directly to the form constructor binds that data and indicates that validation should be performed. Often, though, we need to display an initial form with some of the fields prefilled — for example, an “edit” form. We can do this with the initial keyword argument:  
form = CommentForm(initial={‘sender’: ‘user@example.com’})  
If our form will always use the same default values, we can configure them in the form definition itself:

message = forms.CharField(widget=forms.Textarea(),  
initial=”Replace with your feedback”)

**Creating a Feedback Fom:**

Run this command in the terminal/console:

django-admin startproject feedback

This will create a directory structure like this

Feedback

├── feedback

│   ├── \_\_init\_\_.py

│   ├── settings.py

│   ├── urls.py

│   └── wsgi.py

└── manage.py

Now enter to the project directory by:

cd feedback

Well, now we will create an app named ‘form’:

python manage.py startapp form

This will create the directory structure like this:

feedback

├── feedback

│   ├── \_\_init\_\_.py

│   ├── settings.py

│   ├── urls.py

│   └── wsgi.py

├── form

│   ├── admin.py

│   ├── apps.py

│   ├── \_\_init\_\_.py

│   ├── migrations

│   │   └── \_\_init\_\_.py

│   ├── models.py

│   ├── tests.py

│   └── views.py

└── manage.py

If you know the basics of django, you know what’s going on. If you stuck here, please read the [django official tutorial](https://docs.djangoproject.com/en/1.9/intro/" \t "_blank) first.

## **Enabling the app at settings.py**

Open the **feedback/settings.py** file and add ‘form’ in the INSTALLED\_APPS list, like this:

|  |  |
| --- | --- |
| 33  34  35  36  37  38  39  40  41 | INSTALLED\_APPS = [  'django.contrib.admin',  'django.contrib.auth',  'django.contrib.contenttypes',  'django.contrib.sessions',  'django.contrib.messages',  'django.contrib.staticfiles',  'form',  ] |

Additionally you can change the time zone from settings.py file. I am setting the time zone from ‘UTC’ to ‘Asia/Dhaka’:

110 TIME ZONE = ‘Asia/Dhaka’

## Creating the models

Models are the python classes which contain the information about database. Read more about [models here](https://docs.djangoproject.com/en/1.9/topics/db/models/). For your information, in this tutorial, we are using SQLite as database backend. It is the default, so no further action is required.

In our feedback system, we will receive product based feedback. So there should be products in our database. For simplicity we assume that, our product has only one property, just its name.

And in the feedback form, we want to collect the customer name, email, related product, detailed feedback, is he/she happy or not, and date.

So to design the models, open the **form/models.py** file and add this code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | from django.db import models  class Product(models.Model):  name = models.CharField(max\_length=120)  def \_\_str\_\_(self):  return self.name  class Feedback(models.Model):  customer\_name = models.CharField(max\_length=120)  email = models.EmailField()  product = models.ForeignKey(Product)  details = models.TextField()  happy = models.BooleanField()  date = models.DateField(auto\_now\_add=True)  def \_\_str\_\_(self):  return self.customer\_name |

Now, run this command for making the migrations (database changes):

python manage.py makemigrations

And apply the migrations by this command:

python manage.py migrate

## Enabling and configuring the admin interface

To use the default admin interface provided by django for our models, we need to create the super user first. Run this command to do that:

python manage.py createsuperuser

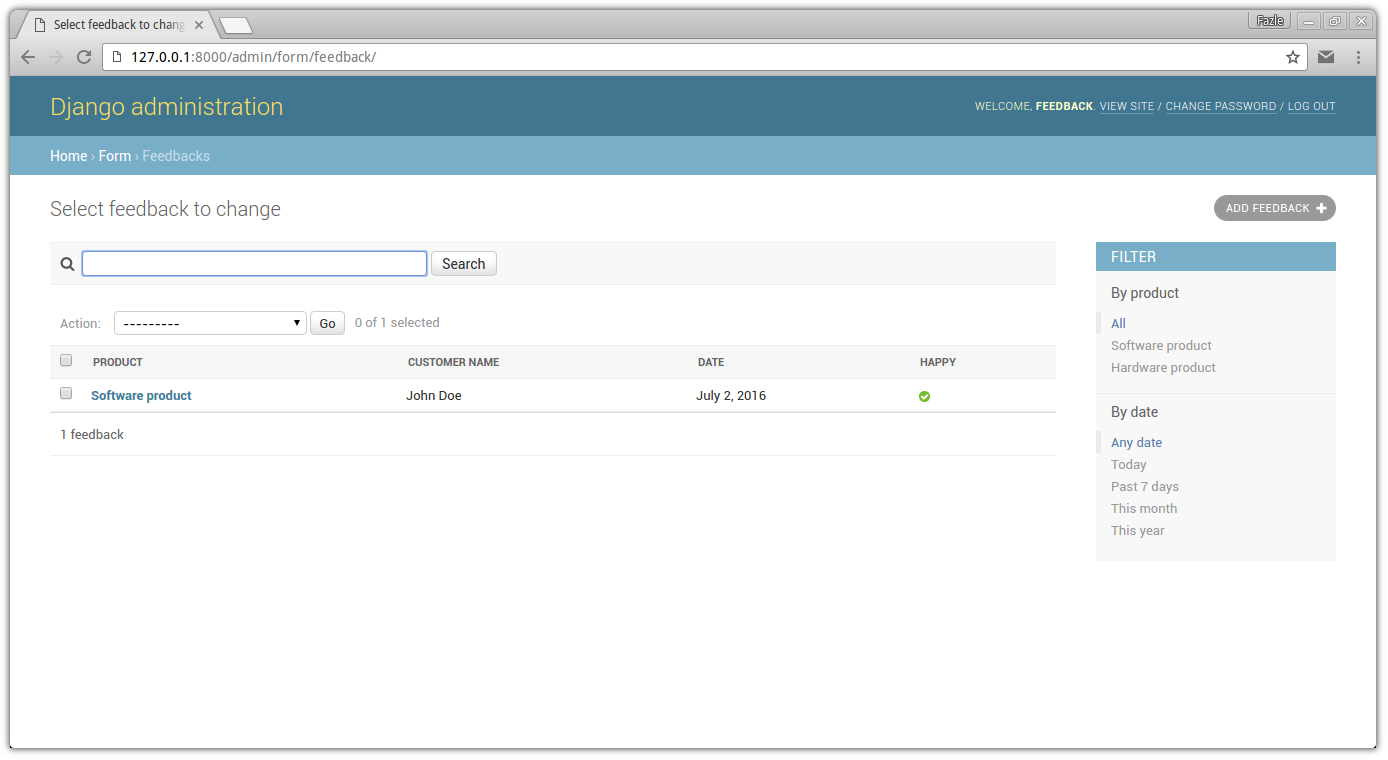
This will prompt for username, email and password. Provide this.

Now to enable and to customize the admin interface of our model, open the **form/admin.py** file and put this code in it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | from django.contrib import admin  from .models import Product, Feedback  class FeedbackAdmin(admin.ModelAdmin):  list\_display = ('product', 'customer\_name', 'date', 'happy',)  list\_filter = ('product', 'date',)  search\_fields = ('product\_\_name', 'details',)  class Meta:  model = Feedback  admin.site.register(Feedback, FeedbackAdmin)  admin.site.register(Product) |

The details about admin site can be found [here](https://docs.djangoproject.com/en/1.9/ref/contrib/admin/). Now run this command to start the development server:python manage.py runserver

Now go to **http://127.0.0.1:8000/admin/** from your browser, log in and add some products and feedback and check that everything works just fine.



## **Creating the view**

First create a new file at **form/forms.py** and put this code in it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | from django import forms  from .models import Feedback  class FeedbackForm(forms.ModelForm):  class Meta:  model = Feedback  exclude = [] |

Learn more about [forms](https://docs.djangoproject.com/en/1.9/topics/forms/) and [model forms](https://docs.djangoproject.com/en/1.9/topics/forms/modelforms/).

Now edit the **form/views.py** in this way:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | from django.shortcuts import render  from .forms import FeedbackForm  def feedback\_form(request):  if request.method == 'POST':  form = FeedbackForm(request.POST)  if form.is\_valid():  form.save()  return render(request, 'form/thanks.html')  else:  form = FeedbackForm()  return render(request, 'form/feedback\_form.html', {'form': form}) |

## **Creating the templates**

Create a folder named ‘templates’ inside the ‘form’ folder. And create another folder named ‘form’ inside the ‘templates’ folder. Now create three empty html file named ‘base.html’, ‘feedback\_form.html’ and ‘thanks.html’ inside the form folder. The directory structure should look like this:

form

├── templates

│   └── form

│   ├── base.html

│   ├── feedback\_form.html

│   └── thanks.html

Now, open the **form/templates/form/base.html** file and add this code to it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <!DOCTYPE html>  <html>  <head>  <title>{% block *title* %}{% endblock %}</title>  </head>  <body>  {% block *body* %}{% endblock %}  </body>  </html> |

Open the **form/templates/form/feedback\_form.html** file and add this code to it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {% extends 'form/base.html' %}  {% block *title* %}Feedback Form{% endblock %}  {% block *body* %}  <form action="{% url 'form:home' %}" method="POST">  {% csrf\_token %}  {{ *form.as\_p* }}  <input type="submit" />  </form>  {% endblock %} |

Finally, open the **form/templates/form/thanks.html** file and add this code to it

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {% extends 'form/base.html' %}  {% block *title* %}Thanks{% endblock %}  {% block *body* %}  <h1>Thanks for your response.</h1>  <p><a href="{% url 'form:home' %}">Response again?</a></p>  {% endblock %} |

You can learn more about [django template language](https://docs.djangoproject.com/en/1.9/ref/templates/language/" \t "_blank).

## **URL configuration**

Create and open the file **form/urls.py** and put this code in it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | from django.conf.urls import url  from . import views  app\_name = 'form'  urlpatterns = [  url(r'^$', views.feedback\_form, name='home'),  ] |

Now, edit the feedback/urls.py if like this way:

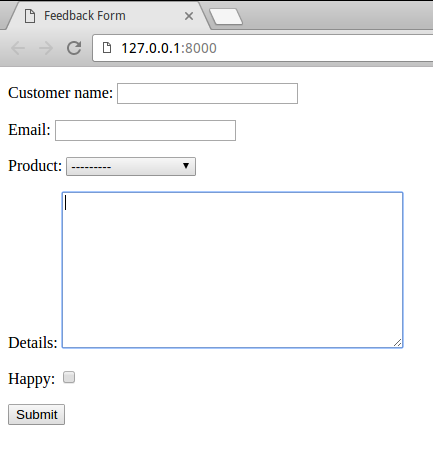
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from django.conf.urls import include, url  from django.contrib import admin  urlpatterns = [  url(r'^', include('form.urls')),  url(r'^admin/', admin.site.urls),  ] |

## Conclusion

All done! Now fire up the django development server by this command:

python manage.py runserver

Now go to \*\*http://127.0.0.1:8000/ \*\*from your browser. You should see like this:



**Processing the Submission The Basics of File submission With Django**

When files are submitted to the server, the file data ends up placed in request.FILES.

It is mandatory for the HTML form to have the attribute enctype="multipart/form-data" set correctly. Otherwise the request.FILES will be empty.

The form must be submitted using the POST method.

Django have proper model fields to handle uploaded files: FileField and ImageField.

The files uploaded to FileField or ImageField are not stored in the database but in the filesystem.

FileField and ImageField are created as a string field in the database (usually VARCHAR), containing the reference to the actual file.

If you delete a model instance containing FileField or ImageField, Django will not delete the physical file, but only the reference to the file.

The request.FILES is a dictionary-like object. Each key in request.FILES is the name from the <input type="file" name="" />.

Each value in request.FILES is an UploadedFile instance.

You will need to set MEDIA\_URL and MEDIA\_ROOT in your project’s settings.py.

**The Basics of File submission With Django**

When files are submitted to the server, the file data ends up placed in request.FILES.

It is mandatory for the HTML form to have the attribute enctype="multipart/form-data" set correctly. Otherwise the request.FILES will be empty.

The form must be submitted using the POST method.

Django have proper model fields to handle uploaded files: FileField and ImageField.

The files uploaded to FileField or ImageField are not stored in the database but in the filesystem.

FileField and ImageField are created as a string field in the database (usually VARCHAR), containing the reference to the actual file.

If you delete a model instance containing FileField or ImageField, Django will not delete the physical file, but only the reference to the file.

The request.FILES is a dictionary-like object. Each key in request.FILES is the name from the <input type="file" name="" />.

Each value in request.FILES is an UploadedFile instance.

You will need to set MEDIA\_URL and MEDIA\_ROOT in your project’s settings.py.

MEDIA\_URL = '/media/'

MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media')

In the development server you may serve the user uploaded files (media) using django.contrib.staticfiles.views.serve() view.

from django.conf import settings

from django.conf.urls.static import static

urlpatterns = [

# Project url patterns...

]

if settings.DEBUG:

urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

To access the MEDIA\_URL in template you must add django.template.context\_processors.media to yourcontext\_processeors inside the TEMPLATES config.

Simple File Upload

Following is a minimal file upload example using FileSystemStorage. Use it just to learn about the flow of the process.

simple\_upload.html

{% extends 'base.html' %}

{% load static %}

{% block content %}

<form method="post" enctype="multipart/form-data">

{% csrf\_token %}

<input type="file" name="myfile">

<button type="submit">Upload</button>

</form>

{% if uploaded\_file\_url %}

<p>File uploaded at: <a href="{{ uploaded\_file\_url }}">{{ uploaded\_file\_url }}</a></p>

{% endif %}

<p><a href="{% url 'home' %}">Return to home</a></p>

{% endblock %}

views.py

from django.shortcuts import render

from django.conf import settings

from django.core.files.storage import FileSystemStorage

def simple\_upload(request):

if request.method == 'POST' and request.FILES['myfile']:

myfile = request.FILES['myfile']

fs = FileSystemStorage()

filename = fs.save(myfile.name, myfile)

uploaded\_file\_url = fs.url(filename)

return render(request, 'core/simple\_upload.html', {

'uploaded\_file\_url': uploaded\_file\_url

})

return render(request, 'core/simple\_upload.html')

File Upload With Model Forms

Now, this is a way more convenient way. Model forms perform validation, automatically builds the absolute path for the upload, treats filename conflicts and other common tasks.

models.py

from django.db import models

class Document(models.Model):

description = models.CharField(max\_length=255, blank=True)

document = models.FileField(upload\_to='documents/')

uploaded\_at = models.DateTimeField(auto\_now\_add=True)

forms.py

from django import forms

from uploads.core.models import Document

class DocumentForm(forms.ModelForm):

class Meta:

model = Document

fields = ('description', 'document', )

views.py

def model\_form\_upload(request):

if request.method == 'POST':

form = DocumentForm(request.POST, request.FILES)

if form.is\_valid():

form.save()

return redirect('home')

else:

form = DocumentForm()

return render(request, 'core/model\_form\_upload.html', {

'form': form

})

model\_form\_upload.html

{% extends 'base.html' %}

{% block content %}

<form method="post" enctype="multipart/form-data">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Upload</button>

</form>

<p><a href="{% url 'home' %}">Return to home</a></p>

{% endblock %}

About the FileField upload\_to Parameter

See the example below:

document = models.FileField(upload\_to='documents/')

Note the upload\_to parameter. The files will be automatically uploaded to MEDIA\_ROOT/documents/.

It is also possible to do something like:

document = models.FileField(upload\_to='documents/%Y/%m/%d/')

A file uploaded today would be uploaded to MEDIA\_ROOT/documents/2016/08/01/.

The upload\_to can also be a callable that returns a string. This callable accepts two parameters, instance andfilename.

def user\_directory\_path(instance, filename):

# file will be uploaded to MEDIA\_ROOT/user\_<id>/<filename>

return 'user\_{0}/{1}'.format(instance.user.id, filename)

class MyModel(models.Model):

upload = models.FileField(upload\_to=user\_directory\_path)

Summary

In this step-by-step tutorial, you learn how to use Python Django to process file upload on HTML forms. We also reviewed how Django model-template-view architectural pattern works.

**Custom Validation Rules**

Imagine we’ve launched our feedback form, and the e-mails have started tumbling in. There’ is just one problem: some of the submitted messages are just one or two words, which isn’t long enough for us to make sense of. We decide to adopt a new validation policy: four words or more, please.

There are a number of ways to hook custom validation into a Django form. If our rule is something we will reuse again and again, we can create a custom field type. Most custom validations are one-off affairs, though, and can be tied directly to the from class. We wanted additional validation on the message field, so we add a clean \_ message() method to our From class:

from django import forms

class ContactForm(forms.Form):

subject = forms.CharField(max\_length=100) email = forms.EmailField(required=False) message = forms.CharField(widget=forms.Textarea)

def clean\_message(self):

message = self.cleaned\_data['message'] num\_words = len(message.split()) if num\_words < 4:

raise forms.ValidationError("Not enough words!") return message

Django’s form system automatically looks for any method whose name starts with clean \_ends with name of a field. If any such methods exists, it’s called during validation.

Specifically, the clean \_ message()method will be called after the default validation logic for a given field (in this case, the validation logic for a required CharField). Because the field data has already been partially processed we pull it out of self\_cleaned\_data. Also, we don’t have to worry about checking that the value exists and is non-empty; that’s done by the default validator.

We naively use a combination of len() and split() to count the number of words entered too few words, we raise a forms Validation Error. The string attached to this exception will be displayed to the user as an item in the error list It’s important that we explicitly return the cleaned 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, then None will be returned, and the original value will be lost.

**Specifying labels**

By default, the labels on Django’s auto-generated form HTML are created by replacing underscores with spaces and capitalizing the first letter – so the label for the email field is "Email". (Sound familiar? It’s the same simple algorithm that Django’ models use to calculate default verbose\_name values for fields. We covered this in chapter 5.)

But, as with Django’s models, we can customize the label for a given field. Just use lables like so:

class ContactForm(forms.Form):

subject = forms.CharField(max\_length=100)

email = forms.EmailField(required=False, **label='Your e-mail address'**) message = forms.CharField(widget=forms.Textarea)

**A custom look and feel**

**Setting a Maximum Length**

One of the most common validation needs is to check that a field is of a certain size.

For good measure, we Contact Form to limit the subject to should improve our max\_length to the CharField, like this:

from django import forms

class ContactForm(forms.Form):

subject = forms.CharField(**max\_length=100**) email =

forms.EmailField(required=False) message =

forms.CharField(widget=forms.Textarea)

An optional min\_length argument is also available.

**Setting Initial Values**

As an improvement to this form, let’s add an initial value for the subject field: “I love your site!”

(A little power of suggestion can’t hurt.) To do this, we can use the initial argument when we create a Form instance:

def contact(request):

if request.method == 'POST': form = ContactForm(request.POST) if form.is\_valid(): cd = form.cleaned\_data send\_mail( cd['subject'], cd['message'], cd.get('email', 'noreply@example.com'),

['siteowner@example.com'], ) return HttpResponseRedirect('/contact/thanks/') else:

form = ContactForm( **initial={'subject': 'I love your site!'}**

) return render(request, 'contact\_form.html', {'form': form})

Now, the subject field will be displayed prepopulated with that kind statement.

Note that there is a difference between passing initial data and passing data that binds the form. The biggest difference is that if you’re just passing initial data, then the form.

Our contact\_ form.html template uses { { form as\_ table } }to display the form, but we can display the form in her other ways to get more granular control over display.

The quickest way to customize forms’ presentation is with CSS. Error lists, in particular, could do with some visual enhancement, and the auto-generated error lists use <ul class = “errorlist”> precisely so that you can target them with CSS. 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 you’ll want to override the default rendering. { { form. as\_ table } } and friends are useful shortcuts while you develop your application, but everything about the way a form is displayed can be overridden, mostly within the template itself, and you’ll probably find yourself doing this. Each field’s widget (<input type="text">, <select>, <textarea>, etc.) can be rendered individually by accessing { { form. fieldname } } in the template, and any errors associated with a field are available as {{ form.fieldname.errors }}. With this in mind, we can construct a custom template for our contact form with the following

template code:

<html>

<head>

<title>Contact us</title>

</head>

<body>

<h1>Contact us</h1>

{% if form.errors %}

<p style="color: red;"> Please correct the error{{ form.errors|pluralize }} below.

</p>

{% endif %}

<form action="" method="post">

<div class="field">

{{ form.subject.errors }}

<label for="id\_subject">Subject:</label>

{{ form.subject }}

</div>

<div class="field">

{{ form.email.errors }}

<label for="id\_email">Your e-mail address:</label>

{{ form.email }}

</div>

<div class="field">

{{ form.message.errors }}

<label for="id\_message">Message:</label>

{{ form.message }}

</div>

<input type="submit" value="Submit">

</form>

</body>

</html>

{{ form.message.errors }} displays a <ul class = “errorlist”> if errors are in 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="field{% if form.message.errors %} errors{% endif %}"> {% if form.message.errors %}

<ul>

{% for error in form.message.errors %}

<li><strong>{{ error }}</strong></li>

{% endfor %}

</ul>

{% endif %}

<label for="id\_message">Message:</label>

{{ form.message }}

</div>

In the case of validation errors, this will add an “errors” class to the containing <div> and display the list of errors in an unordered list.

# Creating forms from models

## ModelForm

*class*ModelForm

If you’re building a database-driven app, chances are you’ll have forms that map closely to Django models. For instance, you might have a BlogComment model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you’ve already defined the fields in your model.

For this reason, Django provides a helper class that lets you create a Form class from a Django model.

For example:

**>>> from** **django.forms** **import** ModelForm

**>>> from** **myapp.models** **import** Article

# Create the form class.

**>>> class** **ArticleForm**(ModelForm):

**...**  **class** **Meta**:

**...**  model = Article

**...**  fields = ['pub\_date', 'headline', 'content', 'reporter']

# 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)

### **Field types**

The generated Form class will have a form field for every model field specified, in the order specified in the fields attribute.

Each model field has a corresponding default form field. For example, a CharField on a model is represented as a CharField on a form. A model ManyToManyField is represented as a MultipleChoiceField. Here is the full list of conversions:

| **Model field** | **Form field** |
| --- | --- |
| [AutoField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.AutoField) | Not represented in the form |
| [BigAutoField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.BigAutoField) | Not represented in the form |
| [BigIntegerField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.BigIntegerField) | [IntegerField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.IntegerField) with min\_value set to -9223372036854775808 and max\_value set to 9223372036854775807. |
| [BinaryField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.BinaryField) | [CharField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.CharField), if [editable](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.Field.editable) is set to True on the model field, otherwise not represented in the form. |
| [BooleanField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.BooleanField) | [BooleanField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.BooleanField), or [NullBooleanField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/" \l "django.forms.NullBooleanField" \o "django.forms.NullBooleanField) if null=True. |
| [CharField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.CharField) | [CharField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.CharField) with max\_length set to the model field’s max\_length and [empty\_value](https://docs.djangoproject.com/en/4.0/ref/forms/fields/" \l "django.forms.CharField.empty_value" \o "django.forms.CharField.empty_value) set to None if null=True. |
| [DateField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.DateField) | [DateField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.DateField) |
| [DateTimeField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.DateTimeField) | [DateTimeField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.DateTimeField) |
| [DecimalField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.DecimalField) | [DecimalField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.DecimalField) |
| [DurationField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.DurationField) | [DurationField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.DurationField) |
| [EmailField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.EmailField) | [EmailField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.EmailField) |
| [FileField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.FileField) | [FileField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.FileField) |
| [FilePathField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.FilePathField) | [FilePathField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.FilePathField) |
| [FloatField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.FloatField) | [FloatField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.FloatField) |
| [ForeignKey](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.ForeignKey) | [ModelChoiceField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.ModelChoiceField) (see below) |
| [ImageField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.ImageField) | [ImageField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.ImageField) |
| [IntegerField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.IntegerField) | [IntegerField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.IntegerField) |
| IPAddressField | IPAddressField |
| [GenericIPAddressField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.GenericIPAddressField) | [GenericIPAddressField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.GenericIPAddressField) |
| [JSONField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.JSONField) | [JSONField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.JSONField) |
| [ManyToManyField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.ManyToManyField) | [ModelMultipleChoiceField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.ModelMultipleChoiceField) (see below) |
| [PositiveBigIntegerField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.PositiveBigIntegerField) | [IntegerField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.IntegerField) |
| [PositiveIntegerField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.PositiveIntegerField) | [IntegerField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.IntegerField) |
| [PositiveSmallIntegerField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.PositiveSmallIntegerField) | [IntegerField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.IntegerField) |
| [SlugField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.SlugField) | [SlugField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.SlugField) |
| [SmallAutoField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.SmallAutoField) | Not represented in the form |
| [SmallIntegerField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.SmallIntegerField) | [IntegerField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.IntegerField) |
| [TextField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.TextField) | [CharField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.CharField) with widget=forms.Textarea |
| [TimeField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.TimeField) | [TimeField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.TimeField) |
| [URLField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.URLField) | [URLField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.URLField) |
| [UUIDField](https://docs.djangoproject.com/en/4.0/ref/models/fields/#django.db.models.UUIDField) | [UUIDField](https://docs.djangoproject.com/en/4.0/ref/forms/fields/#django.forms.UUIDField) |

As you might expect, the ForeignKey and ManyToManyField model field types are special cases:

* ForeignKey is represented by django.forms.ModelChoiceField, which is a ChoiceField whose choices are a model QuerySet.
* ManyToManyField is represented by django.forms.ModelMultipleChoiceField, which is a MultipleChoiceField whose choices are a model QuerySet.

In addition, each generated form field has attributes set as follows:

* If the model field has blank=True, then required is set to False on the form field. Otherwise, required=True.
* The form field’s label is set to the verbose\_name of the model field, with the first character capitalized.
* The form field’s help\_text is set to the help\_text of the model field.
* If the model field has choices set, then the form field’s widget will be set to Select, with choices coming from the model field’s choices. The choices will normally include the blank choice which is selected by default. If the field is required]\=}, this forces the user to make a selection. The blank choice will not be included if the model field has blank=False and an explicit default value (the default value will be initially selected instead).

Finally, note that you can override the form field used for a given model field.