

Django

<https://www.tutorialspoint.com/django/index.htm>

Django is a **web development framework** that assists in building and maintaining quality **web applications**. Django helps **eliminate repetitive tasks** making the development process an **easy and time saving** experience.

It provides a set of tools and features that make it easier to create web applications quickly and efficiently. Django handles common web development tasks like **database management, URL routing, form handling, and user authentication**, allowing developers to focus on building the unique features of the application instead of reinventing the wheel. It follows the **"Don't Repeat Yourself" (DRY)** principle, promoting code **reuse** and maintainability.

Django – Design Philosophies

Loosely Coupled – Django aims to make each element of its stack independent of the others.

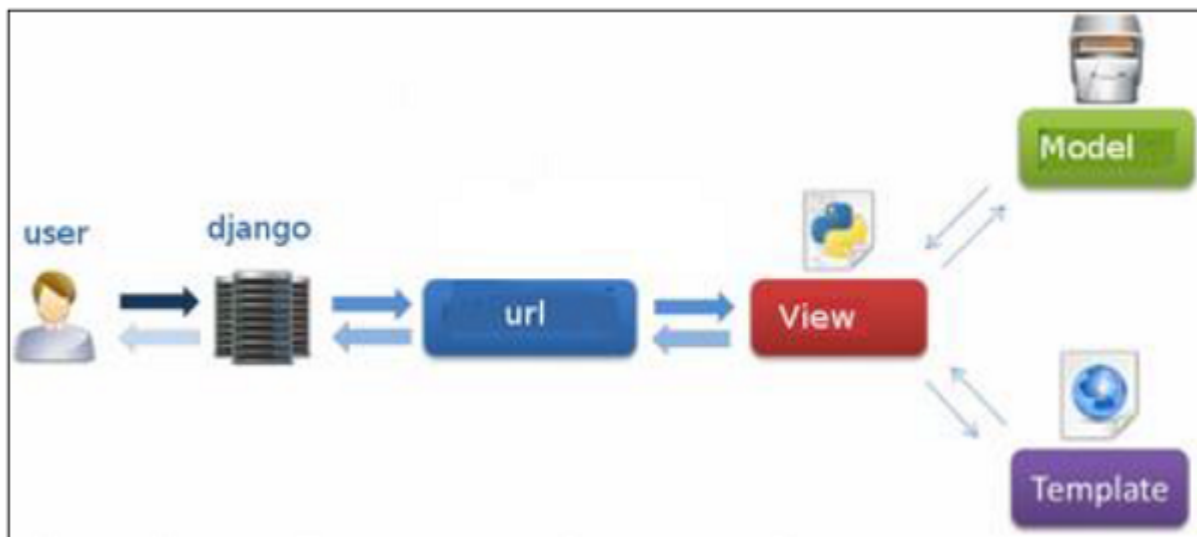
Less Coding – Less code so in turn a quick development.

Don't Repeat Yourself (DRY) – Everything should be developed only in exactly one place instead of repeating it again and again.

Fast Development – Django's philosophy is to do all it can to facilitate hyper-fast development.

Clean Design – Django strictly maintains a clean design throughout its own code and makes it easy to follow best web-development practices.

DJANGO MVC – MVT Pattern



Django Setup

1. Steps to install and create a new project in Django VS code Editor.

- 1) Create a folder and name it with your project name. Ex:- Django_Project_1
- 2) Open VS Code Editor and open the Django_Project_1 folder.
- 3) Open a new terminal in the VS Code Editor.
- 4) Then run the following commands in the terminal, one by one.
 - a) `python3 -m venv myvenv` - - - - Create a virtual environment.
 - b) `source myvenv/bin/activate` - - - - Activating a Virtual environment.
 - c) `pip install django` - - - - Installing Django in a virtual environment.
 - d) `python3 -m pip install --upgrade pip` - - - - Upgrading Python3
 - e) `django-admin startproject django_project_one` - - - - Starting and naming the project “django_project_one”.
 - f) `cd django_project_one` - - - - Changing the directory from the “Django_Project_1” folder to the “django_project_one” project.
 - g) `python3 manage.py runserver` - - To run the server or check the output.

h) To quit the server, press Control + C.

i) `python3 manage.py startapp app_one` – – – To start a new app within the “django_project_one” project.

In Windows

`myenv\Scripts\activate`

`manage.py` – This file is kind of your project local django-admin for interacting with your project via command line (start the development server, sync db...). To get a full list of command accessible via `manage.py` you can use the code

The “myproject” subfolder – This folder is the actual python package of your project. It contains four files –

`__init__.py` – Just for python, treat this folder as package.

`settings.py` – As the name indicates, your project settings.

`urls.py` – All links of your project and the function to call. A kind of ToC of your project.

`wsgi.py` – If you need to deploy your project over WSGI.

`__init__.py` – Just to make sure python handles this folder as a package.

`admin.py` – This file helps you make the app modifiable in the admin interface.

`models.py` – This is where all the application models are stored.

`tests.py` – This is where your unit tests are.

`views.py` – This is where your application views are.

At this stage we have our "myapp" application, now we need to register it with our Django project "myproject". To do so, update `INSTALLED_APPS` tuple in the `settings.py` file of your project (add your app name)

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'myapp',  
)
```

Django - Creating Views

A view function, or “view” for short, is simply a Python function that takes a web request and returns a web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image, etc. Example: You use view to create web pages, note that you need to associate a view to a URL to see it as a web page.

```
from django.http import HttpResponse  
  
def hello(request):  
    text = """<h1>welcome to my app !</h1>"""  
    return HttpResponse(text)
```

```
from django.shortcuts import render
```

```
def hello(request):
```

```
    return render(request, "myapp/template/hello.html", {})
```

Django - URL Mapping

We want to access that view via a URL. Django has its own way for URL mapping and it's done by editing your project url.py file (**myproject/url.py**).

```
from django.urls import path
```

```
from . import views
```

```
urlpatterns = [
```

```
    path('', views.hello_world, name='hello_world'),
```

```
]
```

Django - Index Page

When a new Django project is created with the **startproject** command, the URL **http://localhost:8000/** shows a **default index page**. It shows that the Django installation is done successfully.

```
django-admin startproject myproject
```

```
python manage.py runserver
```

```
Validating models...
```

```
0 errors found
```

```
March 09, 2022 - 12:24:26
```

```
Django version 4.0, using settings 'myproject.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CTRL-BREAK.Quit the server with  
CONTROL-C.
```

Django - Template System

Django makes it possible to separate python and HTML, the python goes in views and HTML goes in templates. To link the two, Django relies on the render function and the **Django Template language**.

The Render Function

Request – The initial request.

The **path** to the template – This is the path relative to the TEMPLATE_DIRS option in the project settings.py variables.

Dictionary of parameters – A dictionary that contains all variables needed in the template. This variable can be created or you can use locals() to pass all local variables declared in the view.

```
def hello(request):  
    today = datetime.datetime.now().date()  
    return render(request, "hello.html", {"today" : today})
```

Filters

{{string|truncatewords:80}} – This filter will truncate the string, so you will see only the first 80 words.

{{string|lower}} – Converts the string to lowercase.

{{string|escape|linebreaks}} – Escapes string contents, then converts line breaks to tags.

Tags

Tags lets you perform the following operations: if condition, for loop, template inheritance and more.

```
<html>
<body>

    Hello World!!!<p>Today is {{today}}</p>
    We are
    {% if today.day == 1 %}

        the first day of month.
    {% elif today.day == 30 %}

        the last day of month.
    {% else %}

        I don't know.
    {%endif%}

</body>
</html>

def hello(request):
    today = datetime.datetime.now().date()

    daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
'Sun']

    return render(request, "hello.html", {"today" : today,
"days_of_week" : daysOfWeek})

<html>
```

```
<body>
```

```
Hello World!!!<p>Today is {{today}}</p>
```

```
We are
```

```
{% if today.day == 1 %}
```

```
the first day of month.
```

```
{% elif today.day == 30 %}
```

```
the last day of month.
```

```
{% else %}
```

```
I don't know.
```

```
{%endif%}
```

```
<p>
```

```
{% for day in days_of_week %}
```

```
{{day}}
```

```
</p>
```

```
{% endfor %}
```

```
</body>
```

```
</html>
```

Block and Extend Tags

A template system cannot be complete without template inheritance. Meaning when you are designing your templates, you should have a main template with holes that the child's template will fill according to his own need, like a page might need a special css for the selected tab.

```
<html>
```



```
<head>
```

```
<title>
```

```
{% block title %}Page Title{% endblock %}
```

```
</title>
```

```
</head>
```

```
<body>
```

```
{% block content %}
```

```
Body content
```

```
{% endblock %}
```

```
</body>
```

```
</html>
```

```
% extends "main_template.html" %}
```

```
{% block title %}My Hello Page{% endblock %}
```

```
{% block content %}
```

```
Hello World!!!<p>Today is {{today}}</p>
```

```
We are
```

```
{% if today.day == 1 %}
```

```
the first day of month.
```

```
{% elif today.day == 30 %}
```

```
the last day of month.
```

```
{% else %}
```

```
I don't know.
```

```
{%endif%}
```

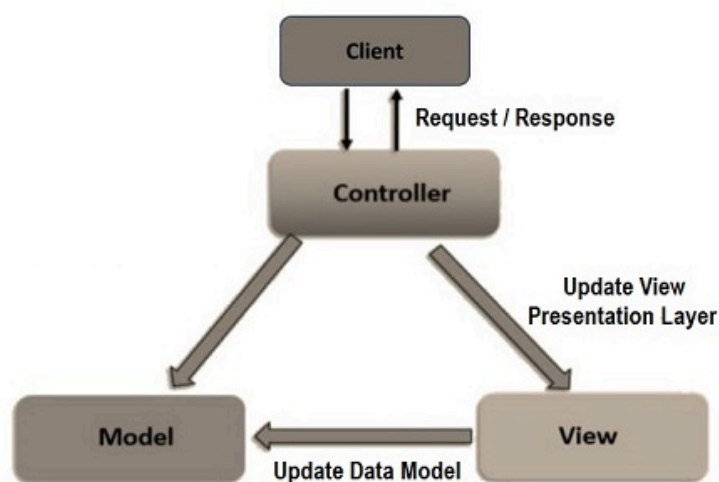
```
<p>
    {% for day in days_of_week %}
    {{day}}
</p>

{% endfor %}
{% endblock %}
```

Django – MVT Architecture

Most of the web frameworks implement the MVC (Model-View-Controller) architecture. Django uses a variation of MVC and calls it the MVT (stands for Model-View-Template) architecture

The MVC Architecture



The Model Layer

The Model is known as the lowest level which means it is responsible for maintaining the data. It handles data.

The model layer is connected to the database. It responds to the controller requests because the controller never talks to the database by itself. The model talks to the database back and forth and then it gives the needed data to the controller.

The model is responsible for data definitions, its processing logic and interaction with the backend database.

The View Layer

The View is the presentation layer of the application. It takes care of the placement and formatting of the result and sends it to the controller, which in turn, redirects it to the client as the application's response.

Data representation is done by the view component. It actually generates the UI or user interface for the user. So, at web applications, when you think of the View component, just think the HTML/CSS part.

Views are created by the data which is collected by the model component but these data aren't taken directly but through the controller, so the view only speaks to the controller.

The Controller Layer

It coordinates with the View layer and the model layer to send the appropriate response back to the client.

The Controller layer receives the request from the client, and forwards it to the model layer. The Model layer updates the data and sends back to the controller. The Controller updates the view and sends back the response to the user.

The MVT Architecture

The Django framework adapts a MVT approach. It is a slight variation of the MVC approach. The acronym MVT stands for Model, View and Template.

Here too, the Model is the data layer of the application. The View is in fact the layer that undertakes the processing logic. The Template is the presentation layer.



The URL Dispatcher

Django's URL dispatcher mechanism is equivalent to Controller in the MVC architecture. The **urls.py** module in the Django project's package folder acts as the dispatcher. It defines the URL patterns. Each URL pattern is mapped with a view function to be invoked when the client's request URL is found to be matching with it.

The URL patterns defined in each app under the project are also included in it.

When the server receives a request in the form of client URL, the dispatcher matches its pattern with the patterns available in the `urls.py` and routes the flow of the application towards its associated view.

The View Function

The View function reads the path parameters, query parameters and the body parameters included in the client's request. It uses this data to interact with the models to perform CRUD operations, if required.

The Model Class

A Model is a Python class. Django uses the attributes of the Model class to construct a database table of a matching structure. Django's Object Relational Mapper helps in performing CRUD operations in an object oriented way instead of invoking SQL queries.

The View uses the data from the client as well as the model and renders its response in the form of a template.

Template

A Template is a web page in which HTML script is interspersed with the code blocks of Django Template Language.

Django's template processor uses any context data received from the View is inserted in these blocks so that a dynamic response is formulated. The View in turn returns the response to the user.

This is how Django's MVT architecture handles the request-response cycle in a web application.

[Django - Add Master Template](#)

The Master Template

A **base class** in any object-oriented language (such as Python) defines attributes and methods and makes them available to the inherited class. In the same way, we need to design a **master template** that provides an overall skeleton for other templates.

The **master template** (sometimes called "base template"), along with the common structure, also marks the dummy blocks. The child template inherits the common structure and overrides the blocks to provide respective contents. Such blocks are marked with "**block – endblock**" construct.

```
{% block block_name %}
```

```
...
```

```
...
```

```
{% endblock %}
```

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

[Django - Admin Interface](#)

Django provides a ready-to-use user interface for administrative activities. We all know how an admin interface is important for a web project. Django automatically generates admin UI based on your project models.

```
$ python manage.py makemigrations
```

```
$ python manage.py migrate
```

```
$ python manage.py createsuperuser
```

```
$ python manage.py runserver
```

That interface will let you administrate Django groups and users, and all registered models in your app. The interface gives you the ability to do at least the "CRUD" (Create, Read, Update, Delete) operations on your models.

[Django Admin - Create User](#)

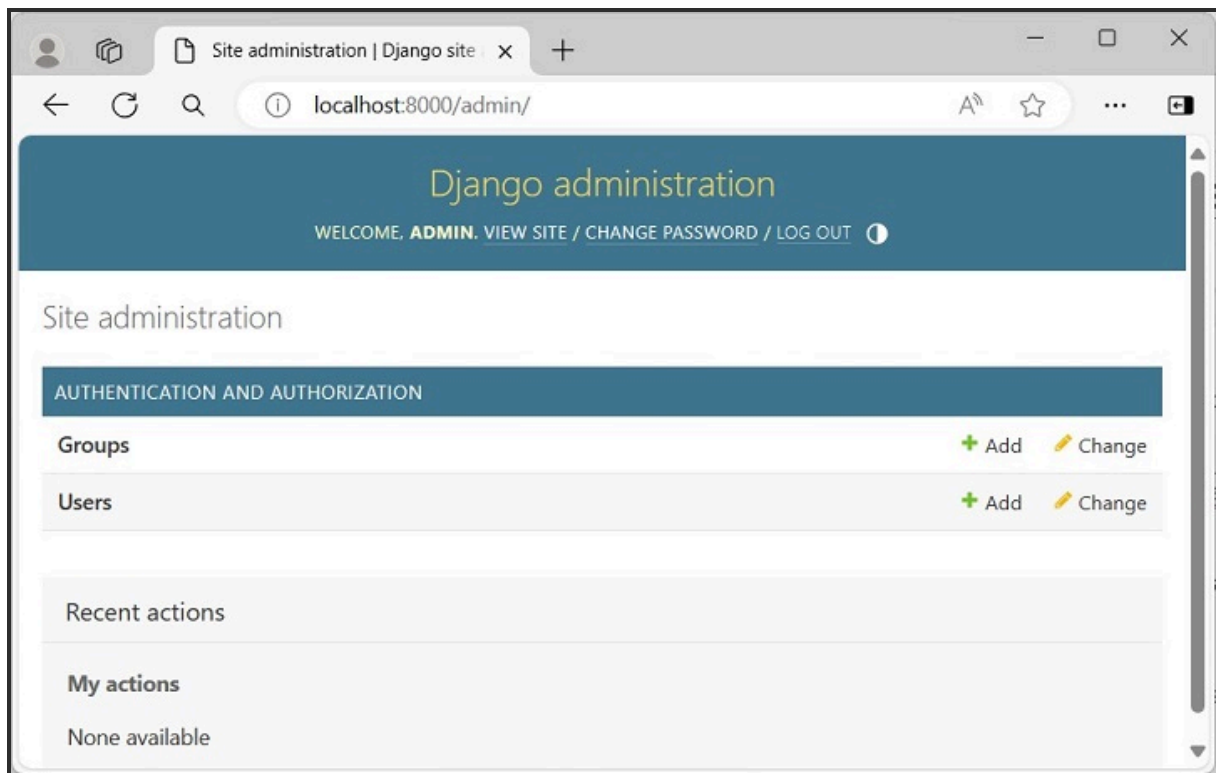
Superuser – A user object that can log into the admin site and possesses permissions to add/change/delete other users as well as perform CRUD operations on all the models in the project, through the admin interface itself.

Staff – The User object has a `is_staff` property. When this property is set to True, the user can login to the Django admin site. The superuser is a staff user by default.

Active – All users are marked as "active" by default. A normal active user (without staff privilege) is not authorized to use the Admin site.

```
python manage.py createsuperuser
```

```
Username: admin
Email address: admin@example.com
Password: ****
Password (again): ****
The password is too similar to the username.
This password is too short. It must contain at least 8
characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```



To create the user programmatically, use the Django shell.

Call the **create_user()** function as follows –

```
>>> from django.contrib.auth.models import User
>>> usr=User.objects.create_user('testusr',
'test@example.com', 'pass123')
>>> usr.is_staff=True
>>> usr.save()
```

Django Admin – Include Models

models.py

```
from django.db import models

# Create your models here.
class Employee(models.Model):
    empno = models.CharField(max_length=20)
    empname = models.CharField(max_length=100)
    contact = models.CharField(max_length=15)
    salary = models.IntegerField()
    joined_date = models.DateField(null=True)
    class Meta:
        db_table = "employee"
```

Shell/Terminal - Migrating the Models. To see and control from the Admin.

```
python manage.py makemigrations myapp
python manage.py migrate
```

admin.py

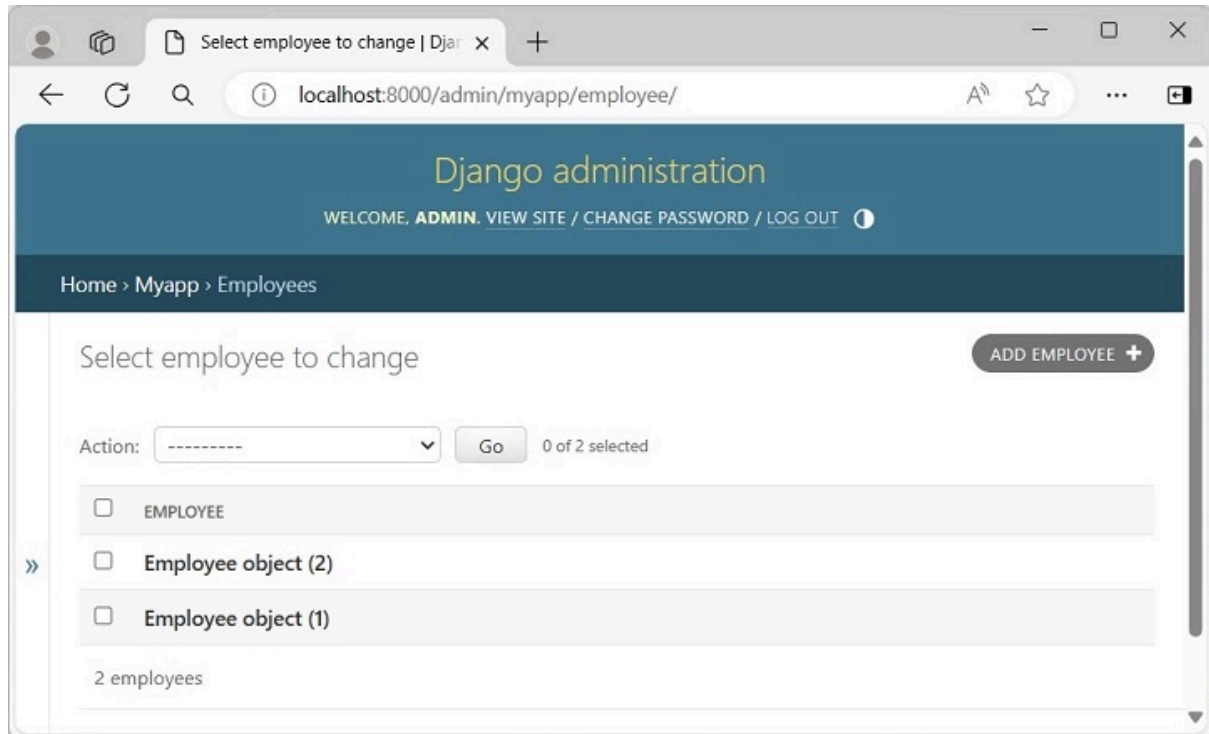
```
from django.contrib import admin

# Register your models here.
from .models import Employee

admin.site.register(Employee)
```


Django Admin – Set Fields to Display

When a model is registered to Django's Admin site, its list of objects is displayed when you click the name of the model.



However, the above page shows the objects in the form Employee object(1), which doesn't reveal the attributes such as the **name** or **contact**, etc. To make the object description more user-friendly, we need to override the **__str__() method** in the **employee model class**.

The **employee class** is rewritten as below to give the alternate string representation of its object.

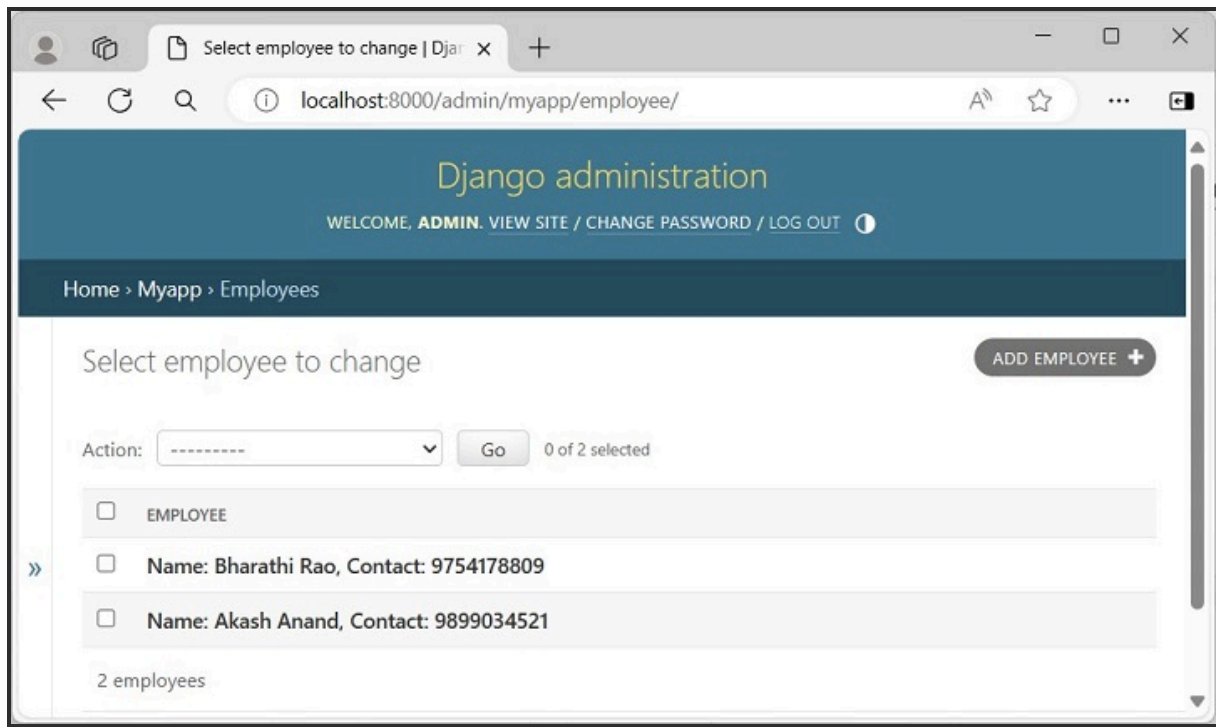
```
from django.db import models

class Employee(models.Model):
    empno = models.CharField(max_length=20)
    empname = models.CharField(max_length=100)
    contact = models.CharField(max_length=15)
    salary = models.IntegerField()
    joined_date = models.DateField(null=True)

    class Meta:
```

```
db_table = "employee"
```

```
def __str__(self):  
    return "Name: {}, Contact: {}".format(self.empname,  
self.contact)
```



```
from django.contrib import admin
```

```
# Register your models here.  
from .models import Employee
```

```
class EmployeeAdmin(admin.ModelAdmin):  
    list_display = ("empname", "contact", "joined_date")  
admin.site.register(Employee, EmployeeAdmin)
```

OR

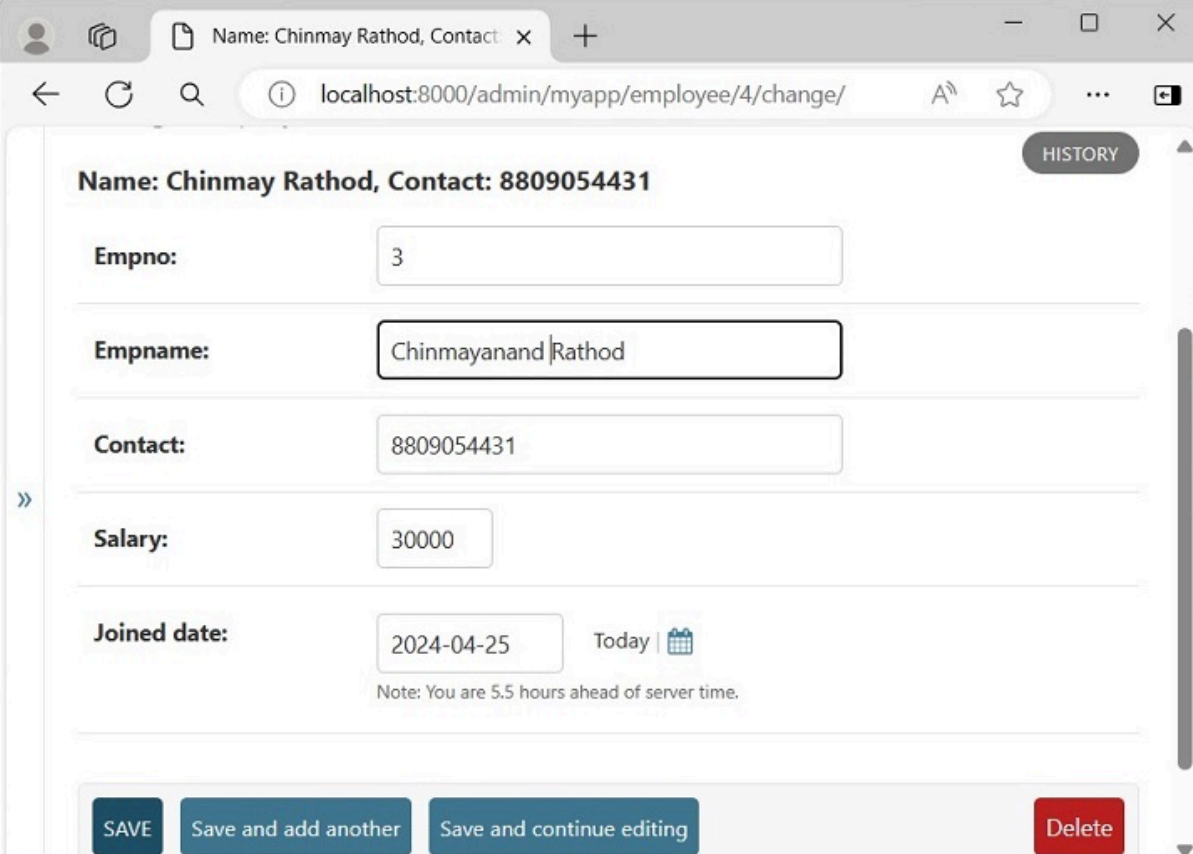
You can also register the model using the **@admin.register()** decorator

```
from django.contrib import admin
```

```
# Register your models here.  
from .models import Employee
```

```
@admin.register(Employee)
class EmployeeAdmin(admin.ModelAdmin):
    list_display = ("empname", "contact", "joined_date")
```

Django Admin - Update Objects



The screenshot shows a web browser window with the URL `localhost:8000/admin/myapp/employee/4/change/`. The page displays the update form for an Employee object. At the top, it shows the current values: "Name: Chinmay Rathod, Contact: 8809054431". Below this, there are several input fields: "Empno:" with the value "3", "Empname:" with the value "Chinmayanand Rathod", "Contact:" with the value "8809054431", "Salary:" with the value "30000", and "Joined date:" with the value "2024-04-25". A "Today" button with a calendar icon is next to the date field. A note at the bottom of the form states: "Note: You are 5.5 hours ahead of server time." At the bottom of the form, there are four buttons: "SAVE", "Save and add another", "Save and continue editing", and "Delete".

Django - Models

A model is a class that represents table or collection in our DB, and where every attribute of the class is a field of the table or collection. Models are defined in the `app/models.py`

Creating a Model

```
from django.db import models
```

```

class Dreamreal(models.Model):

    website = models.CharField(max_length = 50)
    mail = models.CharField(max_length = 50)
    name = models.CharField(max_length = 50)
    phonenumber = models.IntegerField()

    class Meta:
        db_table = "dreamreal"

from myapp.models import Dreamreal
from django.http import HttpResponse

def crudops(request):
    #Creating an entry

    dreamreal = Dreamreal(
        website = "www.polo.com", mail = "sorex@polo.com",
        name = "sorex", phonenumber = "002376970"
    )

    dreamreal.save()

    #Read ALL entries
    objects = Dreamreal.objects.all()
    res = 'Printing all Dreamreal entries in the DB : <br>'

    for elt in objects:
        res += elt.name+"<br>"

    #Read a specific entry:
    sorex = Dreamreal.objects.get(name = "sorex")
    res += 'Printing One entry <br>'
    res += sorex.name

    #Delete an entry
    res += '<br> Deleting an entry <br>'
    sorex.delete()

    #Update
    dreamreal = Dreamreal(
        website = "www.polo.com", mail = "sorex@polo.com",
        name = "sorex", phonenumber = "002376970"

```

```
)
```

```
dreamreal.save()
```

```
res += 'Updating entry<br>'
```

```
dreamreal = Dreamreal.objects.get(name = 'sorex')
```

```
dreamreal.name = 'thierry'
```

```
dreamreal.save()
```

```
return HttpResponse(res)
```

```
from myapp.models import Dreamreal
```

```
def addnew(request):
```

```
    obj = Dreamreal(website="www.polo.com",
```

```
mail="sorex@polo.com", name="sorex", phonenumber="002376970")
```

```
    obj.save()
```

```
    return HttpResponse("<h2>Record Added Successfully")
```

```
from django.urls import path
```

```
from . import views
```

```
urlpatterns = [
```

```
    path("", views.index, name="index"),
```

```
    path("addnew/", views.addnew, name='addnew')
```

```
]
```

Myform.html

```
<html>
```

```
<body>
```

```
    <form action=" ../addnew/" method="post">
```

```
        {% csrf_token %}
```

```
        <p><label for="website">WebSite: </label>
```

```
        <input id="website" type="text" name="website"></p>
```

```
        <p><label for="mail">Email: </label>
```

```
        <input id="mail" type="text" name="mail"></p>
```

```
        <p><label for="name">Name: </label>
```

```
        <input id="name" type="text" name="name"></p>
```

```
        <p><label for="phonenumber">Phone Number: </label>
```

```
        <input id="phonenumber" type="text"
```

```
name="phonenumber"></p>
```

```
        <input type="submit" value="OK">
```

```
</form>
</body>
</html>
```

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR/'templates'],
        .. ..,
    ]
```

```
from django.shortcuts import render
from myapp.models import Dreamreal
```

```
def addnew(request):
    if request.method == "POST":
        ws = request.POST['website']
        mail = request.POST['mail']
        nm = request.POST['name']
        ph = request.POST['phonenumber']
        obj = Dreamreal(website=ws, mail=mail, name=nm,
phonenumber=ph)
        obj.save()
        return HttpResponseRedirect("<h2>Record Added
Successfully</h2>")
    return render(request, "myform.html")
```

The Model Form

Django has a `ModelForm` class that automatically renders a HTML form with its structure matching with the attributes of a model class.

We define a `DreamRealForm` class in `forms.py` file under the app folder that uses the `Dreamreal` model as the basis.

```
from django import forms

from .models import Dreamreal
```

```

class DreamrealForm(forms.ModelForm):

    class Meta:

        model = Dreamreal

        fields = "__all__"

from .forms import DreamrealForm

def addnew(request):

    if request.method == 'POST':

        form = DreamrealForm(request.POST)

        if form.is_valid():

            form.save()

            return HttpResponse("<h2>Book added  
successfully</h2>")

    context={'form' : DreamrealForm}

    return render(request, "myform.html", context)

```

Django – Update Data

Django ORM uses the Active Record pattern for the interaction between a model class and its mapped database table. An instance of the model class corresponds to a single row in the table. Any of the attributes of the object results in updating the corresponding row.

Update Object From Shell

```
python manage.py shell
```

```
>>> from myapp.models import Dreamreal
obj = Dreamreal.objects.filter(pk = 2)
obj.update(f1=v1, f2=v2, . . .)
obj.update(name='example')
```

```
obj = Dreamreal.objects.get(phonenummer = 2376970)
obj.phonenummer = 24642367570
obj.save()
```

```
from django.shortcuts import render
from django.http import HttpResponse
from myapp.models import Dreamreal
```

```
def update(request, pk):
    obj = Dreamreal.objects.get(pk=pk)
    obj.name="admin"
    obj.save()
    return HttpResponse("Update successful")
```

```
from django.urls import path
```

```
from . import views
```

```
from .views import DRCreateView, update
```

```
urlpatterns = [
```

```
    path("", views.index, name="index"),
```

```
    path("addnew/", views.addnew, name='addnew'),
```

```
    path("update/<int:pk>", views.update, name='update'),
```

```
]
```

Modify the update() View Function

```
from django.shortcuts import render
from django.http import HttpResponse
from myapp.models import Dreamreal
```



```

def update(request, pk):
    obj = Dreamreal.objects.get(pk=pk)
    if request.method == "POST":
        ws = request.POST['website']
        mail = request.POST['mail']
        nm = request.POST['name']
        ph = request.POST['phonenumber']
        obj.name = nm
        obj.phonenumber = ph
        obj.save()
        return HttpResponseRedirect("<h2>Record updated  
Successfully</h2>")
    obj = Dreamreal.objects.get(pk=pk)
    context = {"obj":obj}
    return render(request, "myform.html", context)

```

```
<html>
```

```
<body>
```

```
<form action=" ../update/{{ obj.pk }}" method="post">
```

```
{% csrf_token %}
```

```
<p><label for="website">WebSite: </label>
```

```
<input id="website" type="text" value = {{ obj.website  
}} name="website" readonly></p>
```

```
<p><label for="mail">Email: </label>
```

```
<input id="mail" type="text" value = {{ obj.mail }}  
name="mail" readonly></p>
```

```
<p><label for="name">Name: </label>
```

```
<input id="name" type="text" value = {{ obj.name }}  
name="name"></p>
```

```
<p><label for="phonenumber">Phone Number: </label>
```

```
<input id="phonenumber" type="text" value = {{  
obj.phonenumber }} name="phonenumber"></p>
```

```
<input type="submit" value="Update">
```

```
</form>
```

```
</body>
```

```
</html>
```

Django – Update Data

```
python manage.py shell
```

```
>>> from myapp.models import Dreamreal
```

```
row = Dreamreal.objects.get(pk = 1)
```

```
row.delete()
```

```
rows = Dreamreal.objects.filter(name__startswith = 'a')
```

```
rows.delete()
```

```
from django.shortcuts import render
```

```
from django.http import HttpResponse
```

```
from myapp.models import Dreamreal
```

```
def delete(request, pk):
```

```
    obj = Dreamreal.objects.get(pk=pk)
```

```
    obj.delete()
```

```
    return HttpResponse("Deleted successfully")
```

Django – Update Model

The **create()**, **update()** and **delete()** methods perform their respective operations on an already existing table. However, you often need to make changes to the model structure itself, by adding, deleting or altering the attributes of the model. Django's admin interface can be helpful in handling these activities.

migrate – Responsible for applying and unapplying migrations.

makemigrations – Responsible for creating new migrations based on the changes made to the models.

sqlmigrate – Displays the SQL statements for a migration.

showmigrations – Lists a project's migrations and their status

```
python manage.py makemigrations  
Python manage.py migrate
```

Django - Add Static Files

The django template language allows data to be inserted in a web page dynamically. However, a web application also needs certain resources such as images, JavaScript code and style sheets to render the complete web page. These types of files are called **static files**. In a Django application, these files are stored in a static folder inside the application folder.

```
STATIC_URL = 'static/'
```

```
{% load static %}
```

```
<html>
```

```
<body>
```

```
    {% load static %}
```

```
    
```

```
</body>
```

```
</html>
```

```
<link rel="stylesheet" type="text/css" href="{% static  
'style.css' %}">
```

```
<head>
```

```
    {% load static %}
```

```
    <script src = "{% static 'script.js' %}"></script>
```

```
</head>
```

```
from django.shortcuts import render
```

```
def index(request):
```

```
    cities = ['Mumbai', 'New Delhi', 'Kolkata', 'Bengaluru',  
'Chennai', 'Hyderabad']
```

```
    return render(request, "cities.html", {"cities":cities})
```

```
<html>
```

```

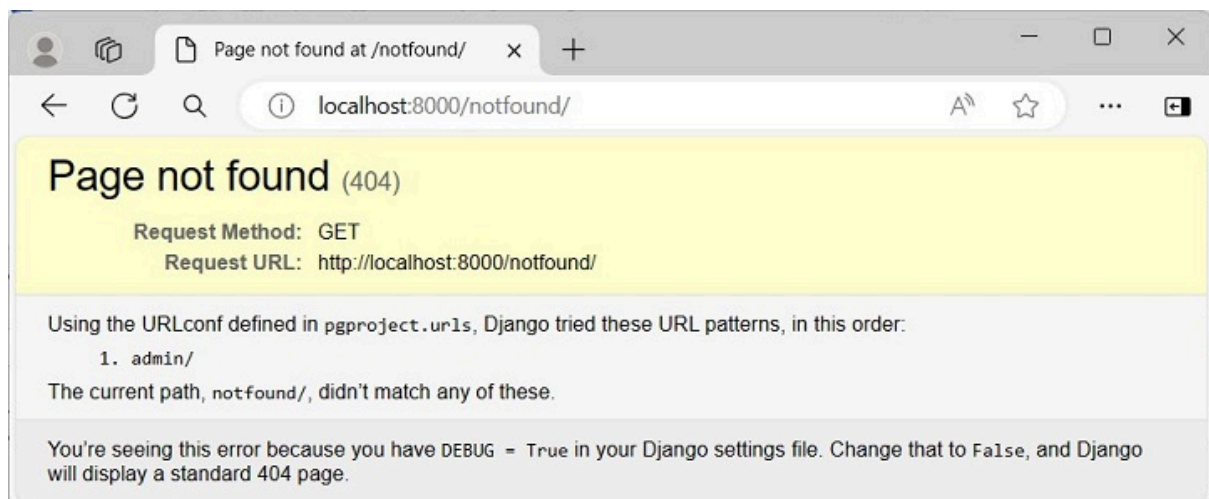
<head>
    {% load static %}
    <link rel="stylesheet" type="text/css" href="{% static
'style.css' %}">
</head>
<body>
    <h2>List of Cities</h2>
    <ul>
        {% for city in cities %}
            <li>{{ city }} </li>
        {% endfor %}
    </ul>
</body>
</html>

```

Django - Page Not Found (404)

The HTTP protocol defines various status codes to indicate different types of HTTP responses. "404" is the HTTP status code that corresponds to the situation when the server cannot find the requested webpage. It is called as "404 error", also known as the "404 Not Found error" or "HTTP 404 Not Found".

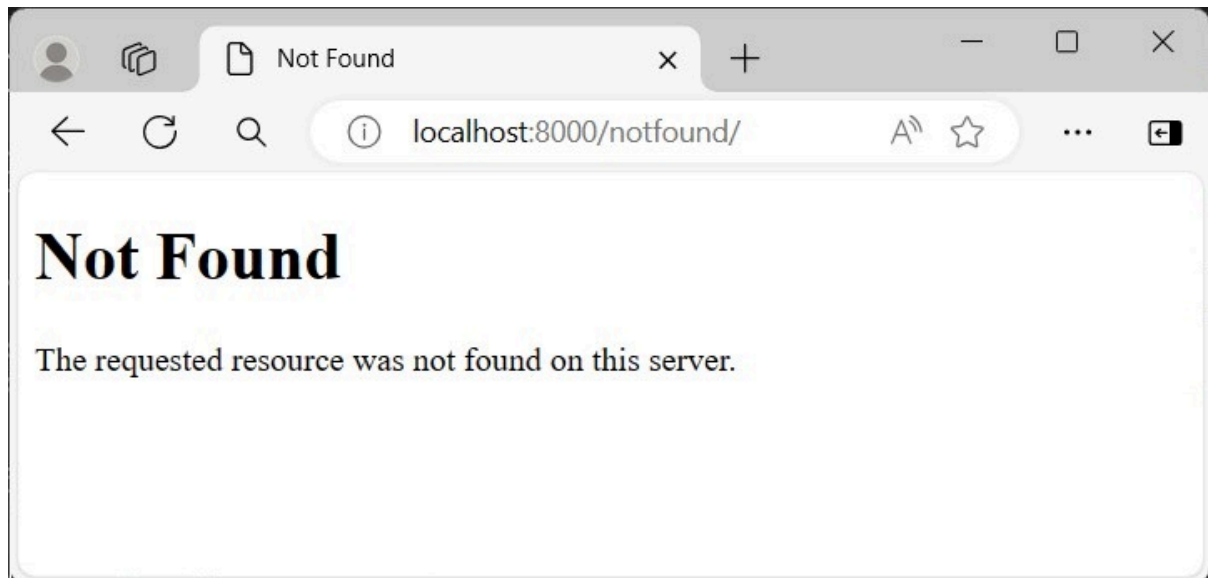
Django's Default Template for 404 Error



Render a Custom Error Page

To render a custom error page, set the `DEBUG` parameter to `FALSE` in the "setings.py" module. Also, you need to specify the list of `ALLOWED_HOSTS` such as `localhost` or a certain domain such as **`https://example.com`**. Set this parameter to `"*"` for any hostname.

```
DEBUG = False
ALLOWED_HOSTS = ["*"]
```



```
from django.shortcuts import render

def handler404(request, exception):
    return render(request, '404handler.html')

from django.contrib import admin
from django.urls import path
from . import views

handler404 = views.handler404

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Django - Page Redirection

Page redirection is needed for many reasons in web application. You might want to redirect a user to another page when a specific action occurs, or basically in case of error. For example, when a user logs in to your website, he is often redirected either to the main home page or to his personal dashboard. In Django, redirection is accomplished using the 'redirect' method.

```
from django.shortcuts import render, redirect
from django.http import HttpResponseRedirect
import datetime

# Create your views here.
def hello(request):
    today = datetime.datetime.now().date()
    daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
    return redirect("https://www.djangoproject.com")

def viewArticle(request, articleId):
    """ A view that display an article based on his ID"""
    text = "Displaying article Number : %s" %articleId
    return redirect(viewArticles, year = "2045", month = "02")

def viewArticles(request, year, month):
    text = "Displaying articles of : %s/%s"%(year, month)
    return HttpResponseRedirect(text)
```

Django - Form Processing

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.

```
from django import forms

class LoginForm(forms.Form):
    user = forms.CharField(max_length = 100)
    password = forms.CharField(widget = forms.PasswordInput())

#-*- coding: utf-8 -*-
```

```

from myapp.forms import LoginForm

def login(request):
    username = "not logged in"

    if request.method == "POST":
        #Get the posted form
        MyLoginForm = LoginForm(request.POST)

        if MyLoginForm.is_valid():
            username = MyLoginForm.cleaned_data['username']
        else:
            MyLoginForm = LoginForm()

    return render(request, 'loggedin.html', {"username" :
username})

```

```

<html>
<body>

    <form name = "form" action = "{% url "myapp.views.login"
%}"
        method = "POST" >{% csrf_token %}

        <div style = "max-width:470px;">
            <center>
                <input type = "text" style = "margin-left:20%;"
                    placeholder = "Identifiant" name =
"username" />
            </center>
        </div>

        <br>

        <div style = "max-width:470px;">
            <center>
                <input type = "password" style =
"margin-left:20%;"
                    placeholder = "password" name = "password"
/>
            </center>
        </div>

```

```

        <br>

        <div style = "max-width:470px;">
            <center>

                <button style = "border:0px;
background-color:#4285F4; margin-top:8%;
                height:35px; width:80%;margin-left:19%;"
type = "submit"
                    value = "Login" >
                        <strong>Login</strong>
                    </button>

            </center>
        </div>

    </form>

</body>
</html>

{% csrf_token %}

<html>

    <body>

        You are : <strong>{{username}}</strong>

    </body>

</html>

```

Using Our Own Form Validation

```

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

from django import forms
from myapp.models import Dreamreal

```



```

class LoginForm(forms.Form):
    user = forms.CharField(max_length = 100)
    password = forms.CharField(widget = forms.PasswordInput())

    def clean_message(self):
        username = self.cleaned_data.get("username")
        dbuser = Dreamreal.objects.filter(name = username)

        if not dbuser:
            raise forms.ValidationError("User does not exist in
our db!")

        return username

```

Django - File Uploading

It is generally useful for a web app to be able to upload files (profile picture, songs, pdf, words.....).

Uploading an Image

Before starting to play with an image, make sure you have the **Python Image Library (PIL)** installed.

Forms.py

```

from django import forms

class ProfileForm(forms.Form):
    name = forms.CharField(max_length = 100)
    picture = forms.ImageField()

```

Models.py

```

from django.db import models

class Profile(models.Model):

```

```
name = models.CharField(max_length = 50)
picture = models.ImageField(upload_to = 'pictures')
```

```
class Meta:
    db_table = "profile"
```

Views.py

```
from myapp.forms import ProfileForm
from myapp.models import Profile

def SaveProfile(request):
    saved = False

    if request.method == "POST":
        #Get the posted form
        MyProfileForm = ProfileForm(request.POST, request.FILES)

        if MyProfileForm.is_valid():
            profile = Profile()
            profile.name = MyProfileForm.cleaned_data["name"]
            profile.picture =
MyProfileForm.cleaned_data["picture"]
            profile.save()
            saved = True
        else:
            MyProfileForm = Profileform()

    return render(request, 'saved.html', locals())
```

HTML/ Template

```
<html>
<body>

    <form name = "form" enctype = "multipart/form-data"
        action = "{% url "myapp.views.SaveProfile" %}" method
= "POST" >{% csrf_token %}

    <div style = "max-width:470px;">
        <center>
            <input type = "text" style = "margin-left:20%;"
                placeholder = "Name" name = "name" />
        </center>
```

```

        </div>
    <br>

    <div style = "max-width:470px;">
        <center>
            <input type = "file" style = "margin-left:20%;"
                placeholder = "Picture" name = "picture" />
        </center>
    </div>

    <br>

    <div style = "max-width:470px;">
        <center>

            <button style =
"border:0px;background-color:#4285F4; margin-top:8%;
                height:35px; width:80%; margin-left:19%;"
type = "submit" value = "Login" >
                <strong>Login</strong>
            </button>

        </center>
    </div>

</form>

</body>
</html>

```

Django - Cookies Handling

To illustrate cookies handling in Django, let's create a system using the login system we created before. The system will keep you logged in for X minute of time, and beyond that time, you will be out of the app.

```

from django.template import RequestContext

def login(request):

```

```

username = "not logged in"

if request.method == "POST":
    #Get the posted form
    MyLoginForm = LoginForm(request.POST)

    if MyLoginForm.is_valid():
        username = MyLoginForm.cleaned_data['username']
    else:
        MyLoginForm = LoginForm()

    response = render_to_response(request, 'loggedin.html',
    {"username" : username},
    context_instance = RequestContext(request))

    response.set_cookie('last_connection',
datetime.datetime.now())
    response.set_cookie('username', datetime.datetime.now())

    return response

```

Django - Sessions

For security reasons, Django has a session framework for cookies handling. Sessions are used to abstract the receiving and sending of cookies, data is saved on server side (like in database), and the client side cookie just has a session ID for identification. Sessions are also useful to avoid cases where the user browser is set to 'not accept' cookies.

'django.contrib.sessions.middleware.SessionMiddleware'

'django.contrib.sessions'

```

def login(request):
    username = 'not logged in'

    if request.method == 'POST':
        MyLoginForm = LoginForm(request.POST)

```

```

        if MyLoginForm.is_valid():
            username = MyLoginForm.cleaned_data['username']
            request.session['username'] = username
        else:
            MyLoginForm = LoginForm()

    return render(request, 'loggedin.html', {"username" :
username})

```

Django - Caching

To cache something is to save the result of an expensive calculation, so that you don't perform it the next time you need it.

```

CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_table_name',
    }
}

```

python manage.py createcachetable

Setting Up Cache in File System

Just add the following in the project settings.py file –

```

CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}

```

Setting Up Cache in Memory

This is the most efficient way of caching, to use it you can use one of the following options depending on the Python binding library you choose for the memory cache –

```
CACHES = {  
    'default': {  
        'BACKEND':  
'django.core.cache.backends.memcached.MemcachedCache',  
        'LOCATION': 'unix:/tmp/memcached.sock',  
    }  
}
```