Django

Django Introduction

Django is a web application framework written in Python programming language. It is based on MVT (Model View Template) design pattern. The Django is very demanding due to its rapid development feature. It takes less time to build application after collecting client requirement.

This framework uses a famous tag line: **The web framework for perfectionists with deadlines.**

By using Django, we can build web applications in very less time. Django is designed in such a manner that it handles much of configure things automatically, so we can focus on application development only.

History

Django was design and developed by Lawrence journal world in 2003 and publicly released under BSD license in July 2005. Currently, DSF (Django Software Foundation) maintains its development and release cycle.

Django was released on 21, July 2005. Its current stable version is 2.0.3 which was released on 6 March, 2018.

Django Versions

Version	Date	Description
0.90	16 Nov 2005	
0.91	11 Jan 2006	magic removal.
0.96	23 Mar 2007	newforms, testing tools.
1.0	3 Sep 2008	API stability, decoupled admin, Unicode.
1.1	29 Jul 2009	Aggregates, transaction based tests.
1.2	17 May 2010	Multiple db connections, CSRF, model validation.
1.3	23 Mar 2011	Timezones, in browser testing, app templates.
1.5	26 Feb 2013	Python 3 Support, configurable user model.
1.6	6 Nov 2013	Dedicated to Malcolm Tredinnick, db transaction management, connection pooling.
1.7	2 Sep 2014	Migrations, application loading and configuration.
1.8 LTS	2 Sep 2014	Migrations, application loading and configuration.
1.8 LTS	1 Apr 2015	Native support for multiple template engines. ***Supported until at least April 2018***
1.9	1 Dec 2015	Automatic password validation. New styling for admin interface.
1.10	1 Aug 2016	Full text search for PostgreSQL. New-style middleware.
1.11LTS	1.11 LTS	Last version to support Python 2.7. ***Supported until at least April 2020***
2.0	Dec 2017	First Python 3-only release, Simplified URL routing syntax, Mobile friendly admin.

Django Popularity

Django is widely accepted and used by various well-known sites. Python-based, Server-Side Web framework that follows the **MTV** architectural pattern. In the last decade, Python's popularity increased by leaps and bounds, which directly affected the high adoption of Django. Besides that, Django offers many pleasant features and is currently one of the main Server-Side Web frameworks.

Key Features:

- It is an enterprise-grade, Server side rendered, **MTV (Model-Template-View) Web framework with additional support for Asynchronous, Reactive programming**. With **Django Admin**, it offers Rapid Application development.
- It is a "Batteries Included" framework and offers everything (e.g., ORM, Middleware, Caching, Security) you need for Rapid Application development with enterprise-grade quality.
- It offers extensibility via pluggable apps where third-party apps can easily be plugged.
- It offers a breakneck development velocity.
- Django works seamlessly with the Python Ecosystem, which is one of the largest ecosystems in the industry.

Django Features

1.Rapid Development

Django was designed with the intention to make a framework which takes less time to build web application. The project implementation phase is a very time taken but Django creates it rapidly.

2.Secure

Django takes security seriously and helps developers to avoid many common security mistakes, such as SQL injection, cross-site scripting, cross-site request forgery etc. Its user authentication system provides a secure way to manage user accounts and passwords.

3.Scalable

Django is scalable in nature and has ability to quickly and flexibly switch from small to large scale application project.

4.Fully loaded

Django includes various helping task modules and libraries which can be used to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds etc.

5.Open Source

Django is an open source web application framework. It is publicly available without cost. It can be downloaded with source code from the public repository. Open source reduces the total cost of the application development.

6.Supported Community

Django is an one of the most popular web framework. It has widely supportive community and channels to share and connect.

Python Virtual Environment

To install Django, first visit to **django official site (https://www.djangoproject.com)** and download django by clicking on the download section. Here, we will see various options to download The Django.

Django requires **pip** to start installation. Pip is a package manager system which is used to install and manage packages written in python. For Python 3.4 and higher versions **pip3** is used to manage packages.

Create New Folder **0_install** and open CMD in there. Change directory by **cd** command.

```
cd 0_install
```

```
python -m venv django2-venv
```

```
.\django2-venv\Scripts\activate
```

```
(django2-venv) python -m pip install django==2.2
```

(1) Django Start Project

Create New Folder **1_start_project** and make project **first_project** in directory.

```
(django2-venv) cd 1_start_project
```

```
(django2-venv) python -m django startproject first_project
```

Project Structures

A Django project contains the following packages and files. The outer directory is just a container for the application.

- **manage.py:** It is a command-line utility which allows us to interact with the project in various ways and also used to manage an application that we will see later on in this tutorial.
- A directory (first_project) located inside, is the actual application package name. Its name is the Python package name which we'll need to use to import module inside the application.
- **__init__.py:** It is an empty file that tells to the Python that this directory should be considered as a Python package.
- **settings.py:** This file is used to configure application settings such as database connection, static files linking etc.
- **urls.py:** This file contains the listed URLs of the application. In this file, we can mention the URLs and corresponding actions to perform the task and display the view.
- **wsgi.py:** It is an entry-point for WSGI-compatible web servers to serve Django project.

Django Run Server

Django project has a built-in development server which is used to run application instantly without any external web server. It means we don't need of Apache or another web server to run the application in development mode.

```
(django2-venv) cd first_project
```

```
(django2-venv) python manage.py runserver
```

Look server has started and can be accessed at localhost with **port 8000**.

The install worked successfully! Congratulations!

(2) Django MySQL Connect

The **settings.py** file contains all the project settings along with database connection details. By default, Django works with **SQLite,** database and allows configuring for other databases as well.

Database connectivity requires all the connection details such as database name, user credentials, hostname drive name etc.

To connect with MySQL, **django.db.backends.mysql** driver is used to establishing a connection between application and database.

Server Stop By Ctrl + C:

Back to main folder and change directory to **2_mysql_connect**.

```
(django2-venv) cd ../..
(django2-venv) cd 2_mysql_connect/first_project
```

```
'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'django_db',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '3306'
    }
```

Create Database Manual

Create database **django_db** with **utf8_general_ci** Collation.

Django Database Migrations

Migration is a way of applying changes that we have made to a model, into the database schema. Django creates a migration file inside the **migration** folder for each model to create the table schema, and each table is mapped to the model of which migration is created.

Django provides the various commands that are used to perform migration related tasks.

- **makemigrations :** It is used to create a migration file that contains code for the tabled schema of a model.
- **migrate :** It creates table according to the schema defined in the migration file.

```
(django2-venv) python manage.py migrate
```

Did you install mysqlclient ?

```
(django2-venv) python -m pip install mysqlclient
```

Django Create Super User

It prompts for login credentials if no password is created yet, use the following command to create a user.

```
(django2-venv) python managen.py createsuperuser
```

```
Username : admin
Email      : admin@gmail.com
Password : superuser
```

It is a Django Admin Dashboard. Here, we can add and update the registered models.

Django Admin Login

Django provides a built-in admin module which can be used to perform CRUD operations on the models. It reads metadata from the model to provide a quick interface where the user can manage the content of the application.

This is a built-in module and designed to perform admin related tasks to the user.

The admin app **(django.contrib.admin)** is enabled by default and already added into **INSTALLED_APPS** section of the settings file.

To access it at browser use '/**admin**/' at a local machine like localhost:8000/admin/.

(3) Django Start App

Django application consists of project and app, it also generates an automatic base directory for the app, so we can focus on writing code (business logic) rather than creating app directories.

The difference between a project and app is, a project is a collection of configuration files and apps whereas the app is a web application which is written to perform business logic.

```
(django2-venv) cd ../..
(django2-venv) cd 3_start_app/first_project
```

```
(django2-venv) python manage.py startapp sample_app
```

the directory structure of the created app, it contains the **migrations** folder to store migration files and model to write business logic.

Initially, all the files are empty, no code is available but we can use these to implement business logic on the basis of the MVC design pattern.

To run this application, we need to make some significant changes which display **hello world** message on the browser.

Open **views.py** file in any text editor and write the given code to it and do the same for **urls.py** file too.

views.py

```
from django.shortcuts import render

# Create your views here.

from django.http import HttpResponse



def hello(request):

    return HttpResponse("<h2> Welcome to Django ! </h2>")
```

urls.py

```
from django.contrib import admin

from django.urls import path

from myapp import views



urlpatterns = [

   path('admin/', admin.site.urls),

   path('hello/', views.hello),

]
```

Open any web browser and enter the URL **localhost:8000/hello**.

Django MVT

The MVT (Model View Template) is a software design pattern. It is a collection of three important components Model View and Template. The Model helps to handle database. It is a data access layer which handles the data.

The Template is a presentation layer which handles User Interface part completely. The View is used to execute the business logic and interact with a model to carry data and renders a template.

A user **requests** for a resource to the Django, Django works as a controller and check to the available resource in URL.

If URL maps, **a view is called** that interact with model and template, it renders a template.

Django responds back to the user and sends a template as a **response**.

(4) Django Model

In Django, a model is a class which is used to contain essential fields and methods. Each model class maps to a single table in the database.

Django Model is a subclass of **django.db.models.Model** and each field of the model class represents a database field (column).

Django provides us a database-abstraction API which allows us to create, retrieve, update and delete a record from the mapped table.

```
(django2-venv) cd ../..
(django2-venv) cd 4_model/first_project
```

Model is defined in **Models.py** file. This file can contain multiple models.

Creating a model **Employee** which has two fields **first_name** and **last_name**.

models.py

```
class Employee(models.Model):
   name = models.CharField(max_length=20)
   address = models.TextField(max_length=50)
```

The **name** and **address** fields are specified as class attributes and each attribute maps to a database column.

The created table contains an auto-created **id field**. The name of the table is a combination of app name and model name that can be changed further.

Register App

After creating model, register model into **INSTALLED_APPS** inside **settings.py.**

```
(django2-venv) python manage.py makemigrations sample_app
(django2-venv) python mange.py migrate
```

Django Model Fields

The fields defined inside the Model class are the columns name of the mapped table. The fields name should not be python reserve words like clean, save or delete etc.

Field Name	Particular
AutoField	It An IntegerField that automatically increments.
BigAutoField	It is a 64-bit integer, much like an AutoField except that it is guaranteed to fit numbers from 1 to 9223372036854775807.
BigIntegerField	It is a 64-bit integer, much like an IntegerField except that it is guaranteed to fit numbers from -9223372036854775808 to 9223372036854775807.
BinaryField	A field to store raw binary data.
BooleanField	A true/false field. The default form widget for this field is a CheckboxInput.
CharField	It is a date, represented in Python by a datetime.date instance.
TimeField	A time, represented in Python by a datetime.time instance.
DateField	It is a date, represented in Python by a datetime.date instance.
DateTimeField	It is used for date and time, represented in Python by a datetime.datetime instance.
DecimalField	It is a fixed-precision decimal number, represented in Python by a Decimal instance.
DurationField	A field for storing periods of time.

EmailField	It is a CharField that checks that the value is a valid email address.
FileField	It is a file-upload field.
FloatField	It is a floating-point number represented in Python by a float instance.
ImageField	It inherits all attributes and methods from FileField, but also validates that the uploaded object is a valid image.
IntegerField	It is an integer field. Values from -2147483648 to 2147483647 are safe in all databases supported by Django.
NullBooleanField	Like a BooleanField, but allows NULL as one of the options.
PositiveIntegerField	Like an IntegerField, but must be either positive or zero (0). Values from 0 to 2147483647 are safe in all databases supported by Django.
SmallIntegerField	It is like an IntegerField, but only allows values under a certain (database-dependent) point.
TextField	A large text field. The default form widget for this field is a Textarea.

Field Options

Each field requires some arguments that are used to set column attributes. For example, CharField requires mac_length to specify varchar database.

Field Options	Particulars
Null	Django will store empty values as NULL in the database.
Blank	It is used to allowed field to be blank.
Choices	An iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field.
Default	The default value for the field. This can be a value or a callable object.
help_text	Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.
primary_key	This field is the primary key for the model.
Unique	This field must be unique throughout the table.

Create a model Employee that contains the following code in **models.py** file.

models.py

```
# Create your models here.
from django.utils import timezone

class Employee(models.Model):
   name = models.CharField(max_length=20)
   address = models.TextField(max_length=50)

   age = models.IntegerField("Age", default=0)
   email = models.EmailField(max_length=50, default='test@gmail.com')
   birthday = models.DateField("Birthday", default=timezone.now)
   image = models.ImageField("Image", default=None)
```

After that apply migration by using the following command.

```
python manage.py makemigrations sample_app
python manage.py migrate
```

It will create a table **sample_app_employee** and browse **phpmyadmin.**

(5) Django Model Admin

One of the most powerful parts of Django is the automatic admin interface. It reads metadata from your models to provide a quick, model-centric interface where trusted users can manage content on your site. The admin's recommended use is limited to an organization's internal management tool. It's not intended for building your entire front end around.

```
(django2-venv) cd ../..
(django2-venv) cd 5_model_admin/first_project
```

admin.py

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

class EmployeeAdmin (admin.ModelAdmin):

    list_display = ['name', 'address', 'age', 'email', 'birthday', 'image']

admin.site.register(Employee, EmployeeAdmin)
```

(6) Django Views

A view is a place where we put our business logic of the application. The view is a python function which is used to perform some business logic and return a response to the user. This response can be the HTML contents of a Web page, or a redirect, or a 404 error.

```
(django2-venv) cd ../..
(django2-venv) cd 6_view/first_project
```

views_calendar.py

```
# Create your views here.
from django.http import HttpResponse
from calendar import HTMLCalendar

def index(request):
   cal = HTMLCalendar().formatmonth(2022, 2)
   return HttpResponse(cal)
```

View calls when gets mapped with URL in **urls.py.**

```
path('calendar/', views_calendar.index),
```

Returning Errors

Django provides various built-in error classes that are the subclass of **HttpResponse** and use to show error message as HTTP response. Some classes are listed below.

Class	Description
class HttpResponseNotModified	It is used to designate that a page hasn't been modified since the user's last request (status code 304).
class HttpResponseBadRequest	It acts just like HttpResponse but uses a 400 status code.
class HttpResponseNotFound	It acts just like HttpResponse but uses a 404 status code.
class HttpResponseNotAllowed	It acts just like HttpResponse but uses a 410 status code.
HttpResponseServerError	It acts just like HttpResponse but uses a 500 status code.

views_404.py

```
# Create your views here.
from django.http import HttpResponse, HttpResponseNotFound

def index(request):
    if True:
        return HttpResponseNotFound('<h1> Page not found </h1>')
    else:
        return HttpResponse('<h1> Page was found </h1>')
```

(7) Django Templates

Django provides a convenient way to generate dynamic HTML pages by using its template system.

A template consists of static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

Why Django Template?

In HTML file, we can't write python code because the code is only interpreted by python interpreter not the browser. We know that HTML is a static markup language, while Python is a dynamic programming language.

Django template engine is used to separate the design from the python code and allows us to build dynamic web pages.

```
(django2-venv) cd ../..
(django2-venv) cd 7_template/first_project
```

Create new app **template_app**.

```
(django2-venv) python mange.py startapp template_app
```

Django Template Configuration

First, create a directory with name **templates** inside the **template_app**.

To configure the templates to system, provide some entries in **settings.py** file.

Find TEMPLATES > DIRS and add os.path.join(BASE_DIR, 'templates').

Create a template **1_index_template.html** inside the created folder.

1_index_template.html

```
<html>
   <head>
      <title> Template </title>
   </head>
   <body>
      <h2> Hello Django Template </h2>
   </body>
</html>
```

Loading Template

To load the template, call get_template() method.

views_template.py

```
# Create your views here.
from django.http import HttpResponse
from django.template import loader

def index(request):
   print('index call me')
   template = loader.get_template('1_index_template.html')
   return HttpResponse(template.render())
```

Django Template Language

Django template uses its own syntax to deal with variable, tags, expressions etc. A template is rendered with a context which is used to get value at a web page.

Variables

Variables associated with a context can be accessed by {{}} (double curly braces). For example, a variable name value is test. Then the following statement will replace name with its value.

2_index_tvaraible.html

```
<html>
   <head>
      <title> Temlate </title>
   </head>
   <body>
      <h2>Employee Information</h2>
      <h3>Name: {{  name }} </h3>
      <h3>Address: {{  address }} </h3>
   </body>
</html>
```

views_tvariable.py

```
# Create your views here.
from django.http import HttpResponse
from django.template import loader

def index(request):
   template = loader.get_template('2_index_tvariable.html')
   employee = {
      'name': 'Aung Aung',
      'address': 'Yangon'
   }
   return HttpResponse(template.render(employee))
```

Tags

In a template, Tags provide arbitrary logic in the rendering process. For example, a tag can output content, serve as a control structure e.g. an "if" statement or a "for" loop, grab content from a database etc.

Tags are surrounded by {% %} braces.

3_index_ttagsif.html

```
{% csrf_token %}
{% if age > 20 %}
        You are adult !
{% else %}
        You are young !
{% endif %}
```

views_ttagsif.py

```
# Create your views here.
from django.http import HttpResponse
from django.template import loader
def index(request):
    template = loader.get_template('3_index_ttagsif.html')
    employee = {
        'name':'Aung Aung',
        'address': 'Yangon',
        'age': 25
    }
    return HttpResponse(template.render(employee))
```

4_index_ttagsfor.html

```
{% csrf_token %}
{% for employee in employee_list %}
   <li> {{ employee }} </li>
{% endfor %}
```

views_ttagsfor.py

```
def index(request):
   employees = ['Kyaw Kyaw', 'Aung Aung', 'Mg Mg', 'Aye Aye']
   return render(request,'4_index_ttagsfor.html',{ "employee_list": employees })
```

5_index_ttable.html

```
<h1> Listing Employees </h1>
{% if employee_list %}
   <table border="1">
      <tr>
         <th>Name</th>
         <th>Address</th>
      </tr>
      {% for employee in employee_list %}
         <tr>
            <td>{{ employee.name }}</td>
            <td>{{ employee.address }}</td>
         </tr>
      {% endfor %}
   </table>
{% else %}
   <p> There are currently no employee to display </p>
{% endif %}
```

views_ttable.py

```
# Create your views here.

class Employee:

    def __init__ (self, name, address, age, email, birthday):

        self.name = name

        self.address = address

        self.age = age

        self.email = email

        self.birthday = birthday


def index(request):

    employee1 = Employee('Kyaw Kyaw', 'Kyeemyindine', 25, 'kyaw@gmail.com', '20-02-2022')

    employee2 = Employee('Aung Aung', 'Ahlone', 18, 'aung@gmail.com', '10-03-2022')

    employees = [ employee1, employee2 ]

    context = { "employee_list": employees }

    return render(request, '5_index_ttable.html', context)
```

Block

Defines a block that can be overridden by child templates.

6_index_ttagsblock.html

```
{% block title %} All Employees {% endblock %}
{% block body %}
   <h1> Listing Employees </h1>
   {% if employee_list %}
      <table border="1">
         <tr>
            <th>Name</th>
            <th>Address</th>
            . . .
         </tr>
         {% for employee in employee_list %}
            <tr>
               <td>{{ employee.name }}</td>
               <td>{{ employee.address }}</td>
               . . .
            </tr>
         {% endfor %}
      </table>
   {% else %}
      <p> There are currently no employee to display </p>
   {% endif %}
{% endblock %}
```

views_ttagsblock.py

```
# Create your views here.
Copy views_ttable.py and change render template name
```

7_index_ttagsextends.html

```
{% extends 'parent_template.html' %}

Copy index_ttagsblock.html
```

parent_template.html

```
<html>
  <head>
    <title> {% block title %} {% endblock %} </title>
  </head>
  <body>
    <u1> This is header </u1>
    <div>
      {% block body %} {% endblock %}
    </div>
    <u1> This is footer </u1>
  </body>
</html>
```

views_ttagsextends.py

```
# Create your views here.

Copy views_ttable.py and change render template name
```

8_index_ttagsinclude.html

```
{% extends 'base.html' %}
Copy index_ttagsblock.html
```

base.html

```
<html>
   <head>
      <title> {% block title %}{% endblock %} </title>
   </head>
   <body>
      {% include 'header.html' %}
      {% block body %}{% endblock %}
      {% include 'footer.html' %}
   </body>
</html>
```

header.html

```
<img src="/static/logo.png"
style="position: relative; right: 30px"
align="right" />
```

footer.html

```
<footer>
   <table class="footer">
      <tr>
         <td>
            <a href="/ttagsextends/"> Home </a>
         </td>
      </tr>
   </table>
</footer>
```

views_ttagsinclude.py

```
# Create your views here.
Copy views_ttable.py and change render template name
```

Create static folder inside **template_app** and add **logo.png** file.

Provide the URL in ***template_app/urls.py***

```
   path('template/', views_template.index),
   path('tvariable/', views_tvariable.index),
   path('ttagsif/', views_ttagsif.index),
   path('ttagsfor/', views_ttagsfor.index),
   path('ttable/', views_ttable.index),
   path('ttagsblock/', views_ttagsblock.index),
   path('ttagsextends/', views_ttagsextends.index),
   path('ttagsinclude/', views_ttagsinclude.index),
```

(8) Django URL

Since Django is a web application framework, it gets user requests by URL locater and responds back. To handle URL, **django.urls** module is used by the framework.

```
(django2-venv) cd ../..
(django2-venv) cd 8_url/first_project
```

Create new app **url_app**.

```
(django2-venv) python mange.py startapp url_app
```

Provide the URL in ***first_project/urls.py***

```
from django.contrib import admin

from django.urls import path, include



urlpatterns = [

    path('admin/', admin.site.urls),

    . . .

    path('url_app/', include('url_app.urls')),

]
```

The view argument is a view function which is used to return a response (template) to the user.

The **django.urls** module contains various functions, **path(route, view, name)** is one of those which is used to map the URL and call the specified view.

Django URL Functions

Commonly used functions for URL handling and mapping.

Name	Description
path(route, view, name=None)	It returns an element for inclusion in urlpatterns.
include(module, namespace=None)	It is a function that takes a full Python import path to another URL conf module that should be "included" in this place.

Create a function **get_employee** in the views.py file. This function will be mapped from the urls.py file.

url_app/views.py

```
# Create your views here.
from django.http import HttpResponse
from sample_app import models


def get_employee(request, employee_id):
    employee = models.Employee.objects.get(id=employee_id)
    name = employee.name
    return HttpResponse(name)
```

Provide the URL in ***url_app/urls.py***

```
from django.urls import path
from url_app import views
urlpatterns = [
            path('get_employee/<int:employee_id>', views.get_employee),
]
```

(9) Django Model Form

It is a class which is used to create an HTML form by using the Model. It is an efficient way to create a form without writing HTML code.

Django automatically does it for us to reduce the application development time. For example, suppose we have a model containing various fields, we don't need to repeat the fields in the form file.

For this reason, Django provides a helper class which allows us to create a Form class from a Django model. First, create a model that contains fields name and other metadata. It can be used to create a table in database and dynamic HTML form.

```
(django2-venv) cd ../..
(django2-venv) cd 9_model_form/first_project
```

Create new app **html_form_app**.

```
(django2-venv) python mange.py startapp html_form_app
```

index_html_form.html

```
<html>
  <head>
    <title> HTML FORM </title>
  </head>
  <body>
    <form method="POST" action="/html_form_app/save_html_form/">
      {% csrf_token %}
        <label> Name </label>
        <input type='text' name='name' required/>
        <br/> <br/>
        <label> Address </label>
        <input type='text' name='address' required/>
        <br/> <br/>
        <label> Age </label>
        <input type='number' name='age' required/>
        <br/> <br/>
        <label> Email </label>
        <input type='email' name='email' required/>
        <br/> <br/>
        <label> Birthday </label>
        <input type='date' name='birthday' required/>
        <br/> <br/>
        <label> Image </label>
        <input type='file' alt='Image' name='image'/>
        <br/> <br/>
      <button type="submit"> Save </button>
    </form>
  </body>
</html>
```

views.py

```
# Create your views here.
from sample_app import models


def index(request):
    employee = models.Employee()
    return render(request,"index_html_form.html", {'employee': employee})


def save(request):
    if request.method == "POST":
        name = request.POST.get('name')
        address = request.POST.get('address')
        age = request.POST.get('age')
        email = request.POST.get('email')
        birthday = request.POST.get('birthday')
        image = request.POST.get('image')
        employee = models.Employee.objects.create(
            name=name,
            address=address,
            age=age,
            email=email,
            birthday=birthday,
            image=image
        )
        employee.save()
        return redirect('/html_form_app/show_html_form')
    else:
        employee = models.Employee()
```

Provide the URL in ***html_form_app/urls.py***

```
from django.urls import path
from html_form_app import views


urlpatterns = [
            path('show_html_form/', views.index),
            path('save_html_form/', views.save),
]
```

Create new app **model_form_app and** Create new python file **forms.py**.

```
(django2-venv) python mange.py startapp model_form_app
```

Creating Form in new created ***froms.py***

```
from django import forms
from sample_app import models

class EmployeeForm(forms.ModelForm):

    class Meta:
        model = models.Employee
        fields = "__all__"
```

Write a view function to load the ModelForm from ***views.py***

```
from django.shortcuts import render, redirect
from model_form_app import forms
from sample_app import models

def index(request):
   form = forms.EmployeeForm()
   return render(request, "index_model_form.html", { 'form': form })

def save(request):
   if request.method == "POST":
      form = forms.EmployeeForm(request.POST, request.FILES)
      if form.is_valid():
         form.save()
         return redirect('/model_form_app/show_model_form')
   else:
      form = forms.EmployeeForm()
   return render(request,"index_model_form.html", { 'form': form })
```

Creating Template in ***index_model_form.html***

```
<html>

   <head>

      <title> MODEL FORM </title>

   </head>

   <body>

      <form method="POST" action="/model_form_app/save_model_form/" enctype="multipart/form-
data">

         {% csrf_token %}

         {{ form.as_p }}

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

      </form>

   </body>

</html>
```

(10) Django Forms

Django provides a Form class which is used to create HTML forms. It describes a form and how it works and appears.

It is similar to the **ModelForm** class that creates a form by using the Model, but it does not require the Model.

Each field of the form class map to the HTML form **<input>** element and each one is a class itself, it manages form data and performs validation while submitting the form.

```
(django2-venv) cd ../..
(django2-venv) cd 10_form/first_project
```

Create new app **form_app**.

```
(django2-venv) python mange.py startapp form_app
```

Creating Form in ***forms.py***

```
from django import forms

class ContactForm(forms.Form):
   username = forms.CharField(max_length=20)
   subject = forms.CharField(max_length=100)
   message = forms.CharField(max_length=100)
```

Passing the context of form into index template in ***views.py***

```
from django.shortcuts import render
from django.http import HttpResponse
from form_app import forms

def index(request):
        if request.method == "POST":
                form = forms.ContactForm(request.POST)
                if form.is_valid():
                        username = form.cleaned_data.get("username")
                        message = "Hello " + username
                        return HttpResponse(message)
        else:
                form = forms.ContactForm()
        return render(request,"contact_form.html", { 'form': form })
```

Creating Template in ***contact_form.html***

```
<html>
   <head>
      <title> CONTACT FORM </title>
   </head>
   <body>
      <form method="POST">
         {% csrf_token %}
         {{ form.as_p }}
         <button type="submit"> Greeting </button>
      </form>
   </body>
</html>
```

Provide the URL in ***form_app/urls.py***

```
from django.urls import path
from form_app import views
urlpatterns = [
        path('create_form/', views.index),
]
```

Commonly used fields and their details are given in the below table.

Name	Class	HTML Input
BooleanField	class BooleanField(**kwargs)	CheckboxInput
CharField	class CharField(**kwargs)	TextInput
ChoiceField	class ChoiceField(**kwargs)	Select
DateField	class DateField(**kwargs)	DateInput
DateTimeField	class DateTimeField(**kwargs)	DateTimeInput
DecimalField	class DecimalField(**kwargs)	NumberInput
EmailField	class EmailField(**kwargs)	EmailInput
FileField	class FileField(**kwargs)	ClearableFileInput
ImageField	class ImageField(**kwargs)	ClearableFileInput

There are other output options though for the <label>/<input> pairs:

{{ form.as_table }} will render them as table cells wrapped in <tr> tags

{{ form.as_p }} will render them wrapped in <p> tags

{{ form.as_ul }} will render them wrapped in <li> tags

(11) Django Shell

With the Python shell with Django, we can access elements in a database, insert new data into a database, update data in a database, etc.

```
(django2-venv) cd ../..
(django2-venv) cd 11_shell/first_project
```

Open shell by command.

```
(django2-venv) python mange.py shell
```

1_create.txt

```
from sample_app import models

employee = models.Employee.objects.create(name='Kyaw Kyaw', address='Yangon')

OR

employee = models.Employee(name='Aung Aung', address='Yangon')

employee.save(force_insert=True)
```

2_read.txt

```
from sample_app import models

GET

employee = models.Employee.objects.get(id=1)
employee.name

FILTER

employee = models.Employee.objects.filter(name='Kyaw Kyaw')
employee[0]

ALL

models.Employee.objects.all()
employee[0]
```

3_update.txt

```
from sample_app import models

models.Employee.objects.filter(id=1).update(name='Aung')

OR

employee_obj = models.Employee.objects.get(id=1)
employee_obj.name = 'AUNG'
employee_obj.save()
```

4_delete.txt

```
models.Employee.objects.filter(name='Kyaw Kyaw').delete()

OR

employee = models.Employee.objects.all()
employee.delete()
```

(12) Django Class Based Generic Views

Writing web applications can be monotonous, because we repeat certain patterns again and again. Django tries to take away some of that monotony at the model and template layers, but web developers also experience this boredom at the view level.

Django's generic views were developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.

We can recognize certain common tasks, like displaying a list of objects, and write code that displays a list of any object. Then the model in question can be passed as an extra argument to the URL conf.

Django ships with generic views to do the following:

- Display list and detail pages for a single object. A single talk page is an example of what we call a "detail" view.
- Present date-based objects in year/month/day archive pages, associated detail, and "latest" pages.
- Allow users to create, update, and delete objects – with or without authorization.

```
(django2-venv) cd ../..
(django2-venv) cd 12_generic_view
```

Create new project **hrms.**

```
(django2-venv) python -m django startproject hrms
```

Create new database **hrms_db** and run migrate with database and create new superuser.

To add images resource, create folder **static** and **templates** for main template file inside **hrms** root folder.

base.html

```
<html>
   <head>
      <title> HRMS </title>
   </head>
   <body>
      {% include 'header.html' %}
      {% block head_block %} {% endblock %}
      {% block body_block %} {% endblock %}
      {% include 'footer.html' %}
   </body>
</html>
```

Add logo.png file at static folder and use from ***header.html***

```
<img src="/static/logo.png"
style="position: relative; right: 30px"
align="right" width="200" height="100" />
```

footer.html

```
<p> This is footer </p>
```

Register new created templates folder and static folder at ***settings.py***

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [
   os.path.join(BASE_DIR,'static')
]
```

Create new app **hr_employees** and register in **settings.py** > INSTALLED_APPS**.**

```
(django2-venv) cd hrms
(django2-venv) python manage.py startapp hr_employees
```

Generic List View

Create Model in ***hr_employees/models.py***

```
from django.utils import timezone

class EmployeeModel(models.Model):
    name = models.CharField(max_length=20, verbose_name='Name')
    birthday = models.DateField(verbose_name='Birthday', default=timezone.now())
    street = models.CharField(max_length=200, verbose_name='Street')
    city = models.CharField(max_length=200, verbose_name='City')
    phone = models.CharField(max_length=20, verbose_name='Phone')
    zip_code = models.CharField(max_length=6, verbose_name='Zip Code')
    image = models.ImageField(upload_to='imgs/', default=None, null=True, blank=True)
```

Create employee list view in ***hr_employee/views.py***

```
from django.views.generic import ListView
from hr_employees import models
from hr_employees import forms

class EmployeeListView(ListView):
    model = models.EmployeeModel
    context_object_name = 'employees_list'
    template_name = 'employee_list.html'
```

hr_employees/templates/employee_list.html

```
{% extends "base.html" %}
{% load static %}
{% block title %} Employees {% endblock %}
{% block body_block %}
<div>
   <div>
      <h2>Employees</h2>
   </div>
  <table border="1">
      <thead>
         <tr>
            <th>No.</th>
            <th>Name</th>
            <th>Birthday</th>
         </tr>
      </thead>
      <tbody>
         {% for employee in employees_list %}
         <tr>
            <td>{{ forloop.counter }}</td>
            <td>{{ employee.name }}</td>
            <td>{{ employee.birthday }}</td>
         </tr>
         {% endfor %}
      </tbody>
   </table>

   <div>
      <p><b>Options:</b></p>
      <p><img src="{% static 'info_ico.png' %}" alt="Details" width="20" height="20">Employee details </p>
      <p><img src="{% static 'del_ico.png' %}" alt="Delete" width="20" height="20">Delete employee </p>
      <p><img src="{% static 'edit_ico.png' %}" alt="Edit" width="20" height="20">Edit employee </p>
   </div>

</div>

{% endblock body_block %}
```

Provide the URL in ***hr_employees/urls.py***

```
from django.urls import path
from hr_employees import views

urlpatterns = [
	path('show_employee/', views.EmployeeListView.as_view(), name='employee_list'),
]
```

Provide the URL in ***hrms/urls.py***

```
from django.contrib import admin

from django.urls import path, include



urlpatterns = [

    path('admin/', admin.site.urls),

    path('hr_employees/', include('hr_employees.urls')),

]

from django.conf import settings

from django.conf.urls.static import static

if settings.DEBUG:

    urlpatterns += static(settings.MEDIA_URL,document_root=settings.MEDIA_ROOT)
```

Register for image upload media folder at ***settings.py***

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

Show all employees in browser **localhost:8000/hr_employees/show_employee/**

Generic Create View

For Create Generic View, Model Form in ***hr_employees/forms.py***

```
from django import forms
from hr_employees import models

class EmployeeCreateForm(forms.ModelForm):
    class Meta:
        model = models.EmployeeModel
        fields = '__all__'
```

Employee create view in ***hr_employee/views.py***

```
from django.views.generic import ListView, CreateView
from django.urls import reverse_lazy

from hr_employees import models
from hr_employees import forms

class EmployeeListView(ListView):
    model = models.EmployeeModel
    context_object_name = 'employees_list'
    template_name = 'employee_list.html'

class EmployeeCreateView(CreateView):
    model = models.EmployeeModel
    form_class = forms.EmployeeCreateForm
    template_name = 'employee_create.html'
```

hr_employees/templates/employee_create.html

```
{% extends 'base.html' %}
{% load static %}

{% block head_block %}
{% endblock %}

{% block body_block %}

<div>
   <h1> Create Employee </h1>
   <form method="POST" action="{% url 'employee_create' %}" enctype="multipart/form-data">
      {% csrf_token %}
      {{ form.as_p }}
      <div>
         <input type="submit" value="Create">
         <a href="javascript:history.back()"> Back </a>
      </div>
   </form>
</div>

{% endblock %}
```

Provide the URL in ***hr_employees/urls.py***

```
from django.urls import path
from hr_employees import views

urlpatterns = [
             path('show_employee/', views.EmployeeListView.as_view(), name='employee_list'),
            path('new_employee/', views.EmployeeCreateView.as_view(), name='employee_create'),
]
```

Generic Update View

For Update Generic View, Model Form in ***hr_employees/forms.py***

```
from django import forms
from hr_employees import models

class EmployeeCreateForm(forms.ModelForm):
    class Meta:
        model = models.EmployeeModel
        fields = '__all__'

class EmployeeUpdateForm(forms.ModelForm):
    class Meta:
        model = models.EmployeeModel
        fields = '__all__'
```

Employee update view in ***hr_employee/views.py***

```
from django.views.generic import ListView, CreateView, UpdateView
from django.urls import reverse_lazy
from hr_employees import models
from hr_employees import forms

class EmployeeListView(ListView):
    . . .
class EmployeeCreateView(CreateView):
    success_url = reverse_lazy("employee_list")
    model = models.EmployeeModel
    form_class = forms.EmployeeCreateForm
    template_name = 'employee_create.html'

class EmployeeUpdateView(UpdateView):
    success_url = reverse_lazy("employee_list")
    model = models.EmployeeModel
    form_class = forms.EmployeeUpdateForm
    context_object_name = "employee"
    template_name = 'employee_update.html'
```

hr_employees/templates/employee_update.html

```
{% extends 'base.html' %}
{% load static %}

{% block head_block %} {% endblock %}

{% block body_block %}
<div>
   <h1> Edit employee </h1>
   <form method="POST" action="{% url 'employee_update' pk=employee.pk %}" enctype="multipart/form-
data">
       {% csrf_token %}
       {{ form.as_p }}
       <div>
          <input type="submit" value="Update">
          <a href="javascript:history.back()"> Back </a>
       </div>
   </form>
</div>
{% endblock %}
```

Provide the URL in ***hr_employees/urls.py***

```
from django.urls import path
from hr_employees import views

urlpatterns = [
            path('show_employee/', views.EmployeeListView.as_view(), name='employee_list'),
            path('new_employee/', views.EmployeeCreateView.as_view(), name='employee_create'),
            path('<int:pk>/edit/', views.EmployeeUpdateView.as_view(), name='employee_update'),
]
```

Add new table head **Options** and new table column for update ***employee_list.html***

```
<td>
        <a href="{% url 'employee_update' pk=employee.pk %}"><img src="{% static 'edit_ico.png' %}"
            alt="Edit" width="20" height="20"></a>
</td>
```

Generic Delete View

Employee delete view in ***hr_employee/views.py***

```
from django.views.generic import ListView, CreateView, UpdateView, DeleteView
. . .

class EmployeeListView(ListView):
    . . .
class EmployeeCreateView(CreateView):
    . . .
class EmployeeUpdateView(UpdateView):
    . . .
class EmployeeDeleteView(DeleteView):
    success_url = reverse_lazy("employee_list")
    model = models.EmployeeModel
    context_object_name = "employee"
    template_name = 'employee_delete.html'
```

hr_employees/templates/employee_delete.html

```
{% extends "base.html" %}
{% block body_block %}
<div>
    <h2> Do you want to delete employee ? </h2>
    <h4> Details </h4>
    <div>
        <ul>
            <li><b>Name: </b> {{ employee.name }} </li>
            <li><b>Birthday: </b> {{ employee.birthday }} </li>
            <li><b>Street: </b> {{ employee.street }} </li>
        </ul>
    </div>
    <form method="POST">
        {% csrf_token %}
        <a href="javascript:history.back()"> Back </a>
        <input type="submit" value="Delete">
    </form>
```

```
</div>
{% endblock %}
```

Provide the URL in ***hr_employees/urls.py***

```
from django.urls import path
from hr_employees import views

urlpatterns = [
        path('show_employee/', views.EmployeeListView.as_view(), name='employee_list'),
        path('new_employee/', views.EmployeeCreateView.as_view(), name='employee_create'),
        path('<int:pk>/edit/', views.EmployeeUpdateView.as_view(), name='employee_update'),
        path('<int:pk>/delete/', views.EmployeeDeleteView.as_view(), name='employee_delete'),
]
```

Add new table head **Options** and new table column for delete ***employee_list.html***

```
<td>
     <a href="{% url 'employee_update' pk=employee.pk %}"><img src="{% static 'edit_ico.png' %}"
        alt="Edit" width="20" height="20"></a>
    <a href="{% url 'employee_delete' pk=employee.pk %}"><img src="{% static 'del_ico.png' %}"
        alt="Edit" width="20" height="20"></a>
</td>
```

Generic Detail View

Employee delete view in ***hr_employee/views.py***

```
from django.views.generic import ListView, CreateView, UpdateView, DeleteView, DetailView
. . .

class EmployeeListView(ListView):
   . . .
class EmployeeCreateView(CreateView):
   . . .
class EmployeeUpdateView(UpdateView):
   . . .
class EmployeeDeleteView(DeleteView):
   . . .
class EmployeeDetailView(DetailView):
   model = models.EmployeeModel
   context_object_name = "employee"
   template_name = 'employee_detail.html'
```

hr_employees/templates/employee_detail.html

```
{% extends "base.html" %}
{% block body_block %}
<div>
   <div>
      <div>
         <h2> Employee details </h2>
      </div>
      <div>
         <ul>
            <li><b>Name: </b> {{ employee.name }} </li>
            <li><b>Birthday: </b> {{ employee.birthday }} </li>
         </ul>
      </div>
      <a href="javascript:history.back()"> Back </a>

   </div>
```

```
</div>
{% endblock %}
```

Provide the URL in ***hr_employees/urls.py***

```
from django.urls import path
from hr_employees import views

urlpatterns = [
        path('show_employee/', views.EmployeeListView.as_view(), name='employee_list'),
        path('new_employee/', views.EmployeeCreateView.as_view(), name='employee_create'),
        path('<int:pk>/edit/', views.EmployeeUpdateView.as_view(), name='employee_update'),
        path('<int:pk>/delete/', views.EmployeeDeleteView.as_view(), name='employee_delete'),
        path('<int:pk>', views.EmployeeDetailView.as_view(), name='employee_detail'),
]
```

Add new table head **Options** and new table column for detail ***employee_list.html***

```
<td>
     <a href="{% url 'employee_update' pk=employee.pk %}"><img src="{% static 'edit_ico.png' %}"
         alt="Edit" width="20" height="20"></a>
    <a href="{% url 'employee_delete' pk=employee.pk %}"><img src="{% static 'del_ico.png' %}"
         alt="Edit" width="20" height="20"></a>
    <a href="{% url 'employee_detail' pk=employee.pk %}"><img src="{% static 'info_ico.png' %}"
        alt="Details" width="20" height="20"></a>
</td>
```

(13) Django Bootstrap

Bootstrap is one of the most popular front-end frameworks out there. It contains some amazing CSS classes for UI development. Bootstrap has pre-defined CSS files and JavaScript code, which you can link with HTML files. Those CSS files contain classes that can be directly used on HTML elements.

```
(django2-venv) cd ../..
(django2-venv) cd 13_bootstrap
(django2-venv) python -m pip install django-bootstrap4
```

Bootstrap CDN (https://getbootstrap.com/)

Bootstrap Content Delivery Network Method will require internet access for every request. Add <link> and <script> in base.html to use bootstrap in our project.

```
<html>
   <head>
      <title> HRMS </title>
      <link rel="stylesheet"
              href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>
      <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"></script>
   </head>
   <body>
      {% include 'header.html' %}
      {% block head_block %} {% endblock %}
      {% block body_block %} {% endblock %}
      {% include 'footer.html' %}
   </body>
</html>
```

Add nav bar in ***header.html***

```
<nav class="navbar navbar-expand-sm bg-primary navbar-dark sticky-top">

  <a class="navbar-brand" href="#"> Human Resource Management System </a>

  <div class="navbar-collapse justify-content-end">
    <ul class="navbar-nav">
      <li class="nav-item">
        <a class="nav-link" href="/petclinic/">Home</a>
      </li>
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdownMenuLink"
            data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
          Employees
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
          <a class="dropdown-item" href="{% url 'employee_list' %}">Employee List</a>
          <a class="dropdown-item" href="{% url 'employee_create' %}">Add Employee</a>
        </div>
      </li>
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdownMenuLink"
            data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
          Contracts
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
          <a class="dropdown-item" href="{% url 'employee_list' %}">Contract List</a>
          <a class="dropdown-item" href="{% url 'employee_create' %}">Add Contract</a>
        </div>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="/petclinic/error/">Error</a>
      </li>
    </ul>
  </div>
</nav>
```

Add container class and table design in ***employee_list.html***

```
. . .
<div class="container">
   <div>
      <h2>Employees</h2>
   </div>

  <table class="table table-bordered table-hover">
      <thead class="thead-light">
         <tr>
            <th scope="col">Options</th>
            <th scope="col">No.</th>
         </tr>
      </thead>
      . . .
```

Use bootstrap4 form features in ***employee_create.html***

```
{% extends 'base.html' %}
{% load bootstrap4 %}
{% load static %}

{% block head_block %} {% endblock %}

{% block body_block %}

<div class="container">
   <h1> Create Employee </h1>
   <form method="POST" action="{% url 'employee_create' %}" enctype="multipart/form-data">
      {% csrf_token %}
      {% bootstrap_form form %}
      <div class="text-center">
         <input type="submit" value="Create" class="btn-action btn btn-primary btn-large">
         <a href="javascript:history.back()" class="btn btn-dark"> Back </a>
      </div>
   </form>
</div>

{% endblock %}
```

Use bootstrap4 form features in ***employee_update.html***

```
{% extends 'base.html' %}
{% load bootstrap4 %}
{% load static %}

{% block head_block %} {% endblock %}

{% block body_block %}
<div class="container">
   <h1> Edit employee </h1>
   <form method="POST" action="{% url 'employee_update' pk=employee.pk %}" enctype="multipart/form-
data">
       {% csrf_token %}
       {% bootstrap_form form %}
       <div class="text-center">
           <input type="submit" value="Update" class="btn-action btn btn-primary btn-large">
           <a href="javascript:history.back()" class="btn btn-dark"> Back </a>
       </div>
   </form>
</div>
{% endblock %}
```

Add container class and change button design in employee_delete.html and employee_detail.html

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-light">Light</button>
<button type="button" class="btn btn-dark">Dark</button>
<button type="button" class="btn btn-link">Link</button>
```

(14) Django Widget

A widget is Django's representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.

```
(django2-venv) cd ../..
(django2-venv) cd 14_widget
(django2-venv) python -m pip install django-flatpickr
```

For date picker widget, add widget attribute in ***hr_employees/forms.py***

```
from django import forms
from hr_employees import models
from flatpickr import DatePickerInput

class EmployeeCreateForm(forms.ModelForm):
   class Meta:
      model = models.EmployeeModel
      fields = '__all__'
      widgets = {
         'birthday': DatePickerInput(options = { "dateFormat": "d.m.y",}),
      }
```

Declare form.media in head block of ***base.html, employee_create.html***

```
{% block head_block %} {{ form.media }} {% endblock %}
```

Must be add datepicker in employee_update.html.

(15) Django User Access Right

LoginRequiredMixin is rather simple and is generally the first inherited class in any view. SuperuserRequiredMixin, another permission-based mixin. This is specifically for requiring a user to be a superuser. Comes in handy for tools that only privileged users should have access to.

```
(django2-venv) cd ../..
(django2-venv) cd 15_access_right
(django2-venv) python -m pip install django-braces==1.13.0
```

Create login form in ***hrms/templates/login.html***

```
{% extends "base.html" %}
{% load bootstrap4 %}

{% block body_block %}
<div class="container">
   <h1 class="text-center"> Welcome HRMS </h1>
   <form method="POST">
      {% csrf_token %}
      {% bootstrap_form form %}
      <input type="submit" value="Sign In" class="btn btn-success btn-block">
   </form>
</div>

{% endblock body_block %}
```

Provide the URL in ***hrms/urls.py***

```
path('login/', auth_views.LoginView.as_view(template_name="login.html"), name="login"),
```

Create normal user and super user access right in ***hr_employees/views.py***

```
. . .
from braces.views import SuperuserRequiredMixin, LoginRequiredMixin
. . .
class EmployeeListView(LoginRequiredMixin, ListView):
    login_url = reverse_lazy('login')
    . . .
class EmployeeCreateView(LoginRequiredMixin, CreateView):
    login_url = reverse_lazy('login')
    . . .
class EmployeeDeleteView(SuperuserRequiredMixin, LoginRequiredMixin, DeleteView):
    login_url = reverse_lazy('login')
    . . .
class EmployeeDetailView(LoginRequiredMixin, DetailView):
    login_url = reverse_lazy('login')
    . . .
class EmployeeUpdateView(SuperuserRequiredMixin, LoginRequiredMixin, UpdateView):
    login_url = reverse_lazy('login')
    . . .
```

Hide update icon and delete icon from ***employee_list.html***

```
{% if user.is_superuser %}
     <a href="{% url 'employee_delete' pk=employee.pk %}"> <img src="{% static 'del_ico.png' %}"
          alt="Delete" width="20" height="20"></a>
     <a href="{% url 'employee_update' pk=employee.pk %}"><img src="{% static 'edit_ico.png' %}"
          alt="Edit" width="20" height="20"></a>
{% endif %}
. . .
{% if user.is_superuser %}
       <p><img src="{% static 'del_ico.png' %}" alt="Delete" width="20" height="20">Delete employee </p>
       <p><img src="{% static 'edit_ico.png' %}" alt="Edit" width="20" height="20">Edit employee </p>
{% endif %}
```

(15) Django QuerySets

QuerySet can be constructed, filtered, sliced, and generally passed around without actually hitting the database. No database activity actually occurs until you do something to evaluate the queryset.

```
(django2-venv) cd ../..
(django2-venv) cd 16_queryset
```

Create search feature in ***employee_list.html***

```
<div style="float: right;">
      <form method="GET" class="form-inline">
          <div class="input-group">
              <input type="search" name="search_query" placeholder="search" class="form-control" required>
              <select name="search_type" style="margin-left:10px;" class="search-select" >
                  <option value="name" selected>Name</option>
                  <option value="street" selected>Street</option>
              </select>
              <div style="margin-left:10px;">
                  <button type="submit" class="btn btn-primary" >Search</button>
              </div>
          </div>
     </form>
</div>
<table ...>
```

Add search function in ***hr_employees/views.py***

```
from django.db.models import Q

class EmployeeListView(LoginRequiredMixin, ListView):
  . . . .
  def get_queryset(self):
    qs = super().get_queryset()
    sq = self.request.GET.get("search_query")
    search_type = self.request.GET.get("search_type")

    if sq is not None:
      if search_type == "name":
        qs = qs.filter(Q(name__icontains=sq))
      elif search_type == "street":
        qs = qs.filter(Q(street__icontains=sq))

    return qs
```

(16) Django Foreign Keys

In **database design many to one** (or one to many) refers to a relationship between one or more entities where **a single entity can have many entities connected**.

```
(django2-venv) cd ../..
(django2-venv) cd 17_foreign_keys
```

Many-to-one Relationships

The many-to-one relationship is known as foreign key relationship in Django, many objects in one model may be related to one object in another model. A employee can have more than one contract working on it. Create Many to one relationship, to compare or join between two entities **Employee** and **Contract**.

```
(django2-venv) python manage.py startapp hr_contracts
```

Create contract model in ***hr_contracts/models.py***

```
from django.utils import timezone

class ContractModel(models.Model):
    name = models.CharField(max_length=20, verbose_name='Name')
    start_date = models.DateField(verbose_name='Start Date', default=timezone.now())
    end_date = models.DateField(verbose_name='End Date', default=timezone.now())

    def __str__(self):
        return self.name
```

Join Contract Model and Employee Model in ***hr_employees/models.py***

```
from django.utils import timezone
from hr_contracts import models as cm

class EmployeeModel(models.Model):
    name = models.CharField(max_length=20, verbose_name='Name')
    . . .
    contract_id = models.ForeignKey(cm.ContractModel, on_delete=models.CASCADE, default=None)
```

Many-to-many Relationship

Many (zero or more) objects in one model can be related to zero or more objects in another model. Create Many to many relationships between a contract and a job, a contract can have zero or more jobs. In the same way, a job can have by 0 or more contracts.

```
(django2-venv) python manage.py startapp hr_jobs
```

Create job model in ***hr_jobs/models.py***

```
class JobModel(models.Model):
    name = models.CharField(max_length=20, verbose_name='Name')

    def __str__(self):
        return self.name
```

Join Job Model and Contract Model in ***hr_contracts/models.py***

```
. . .
from hr_jobs import models as jm

class ContractModel(models.Model):
    name = models.CharField(max_length=20, verbose_name='Name')
    . . .
    job_id = models.ManyToManyField(jm.JobModel)
```