Introduction

Django is a web application framework written in Python programming language. It is a MVT (Model View Template) based framework. The concept behind to build Django was to develop a framework that can meet fast-moving newsroom deadlines while satisfying the tough requirements of clients.

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 design in such a manner that it takes care of much of the hassle of web development, so we can focus on developing application 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 it?s development and release cycle.

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

Django Version History

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Date** |
| 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. |
| 1.11 LTS | 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. |

Popularity

Django is widely accepted and used by various well-known sites such as:

* Instagram
* Mozilla
* Disqus
* Pinterest
* Bitbucket
* The Washington Times

Features of Django

* Rapid Development
* Secure
* Scalable
* Fully loaded
* Versatile
* Open Source
* Vast and Supported Community

Rapid Development

Django was designed with the intention to help developers travel from a concept to a finished project as quickly as possible. When speaking about speed one has to consider performance. Django partakes in lower memory consumption and fewer demands on the database/network, becoming much less of a burden than many other frameworks out there.

Secure

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

Scalable

Some of the busiest sites on the planet use Django's ability to quickly and flexibly scale to meet the heaviest traffic demands.

Fully loaded

Django includes dozens of extras you can use to handle common Web development tasks. Django takes care of user authentication, content administration, site maps, RSS feeds, and many more tasks-right out of the box.

Versatile

Companies, organizations and governments have used Django to build all sorts of things-from content management systems to social networks to scientific computing platforms.

**Open Source**

**Vast and Supported Community**

Django Installation

Django installation process is pretty easy, we just need to visit **django official site (https://www.djangoproject.com)** and click on the download section. Here, we can download and install Django in our local system.

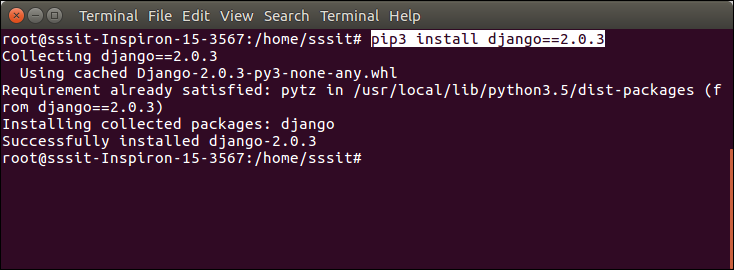
In this tutorial, we are installing Django in Ubuntu operating system.

The whole process installation process is described below. Before installing make sure we have **pip installed**.

Pip is a package manager system which is used to install and manage packages written in python. For Python 3.4 and later **pip3** is used to manage packages.

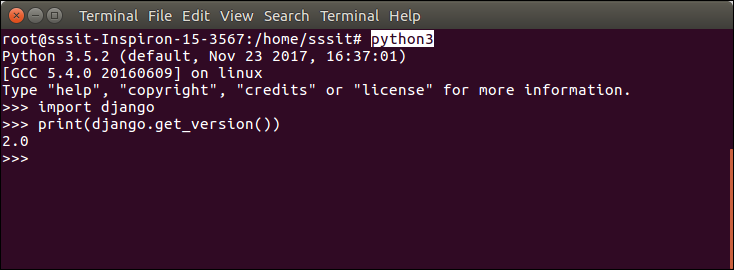
Here, we are installing using pip3, the command is given below.

1. $ pip3 install django==2.0.3



Verify Django Installation

Open terminal and write **python3** and press enter. It will enter into python shell where we can verify django installation.



Look at the Django version given by the print method of python.

Well, this is whole process to install Django in Ubuntu operating system that we be used to create django web applications.

Django Project

In previous topic, we have installed Django successfully. Now, we will see step by step process to creating Django application.

To create Django project, use the following command.

1. $ django-admin startproject projectname

Django Project Example

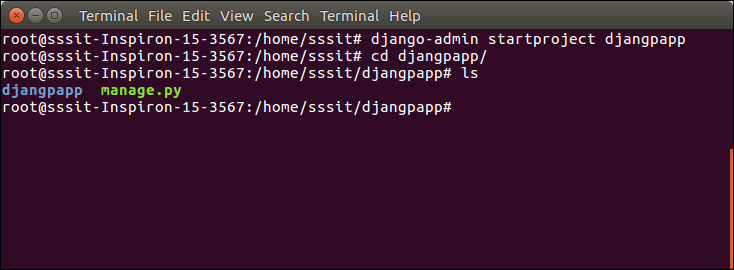
1. $ django-admin startproject djangpapp



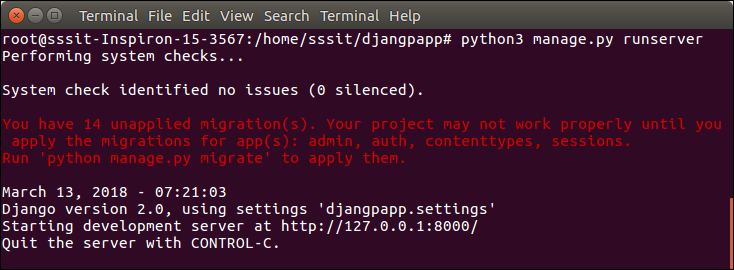
Locate into the Project

Change the directory and list out the packages and files inside of django project.

1. cd djangpapp



We can also use **tree** command to view tree structure of the application. This is a utility command, if it is not present then can be downloaded via **apt-get install tree** command.



A Django package contains the following packages and files. The outer directory is just a container for the application. We can rename it further.

* **manage.py:** It is a command-line utility which allows us to interact with project in various ways and also used to manage application that we will see later on in this tutorial.
* A directory (djangpapp) located inside, is the actual application package name. Its name is the Python package name which we'll need to use to import anything 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 provide 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.

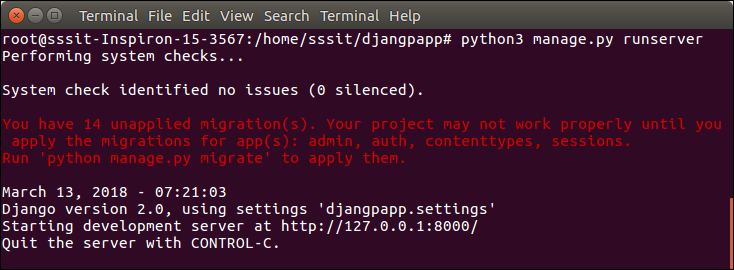
Initially, this is a default draft of the application that contains all required files and folders

Running the Django Project

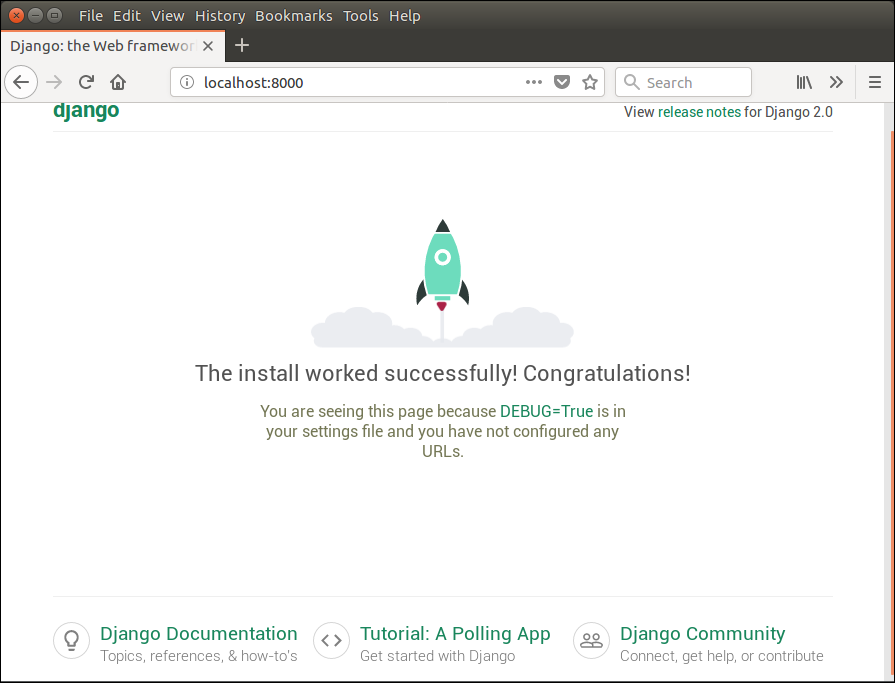
Django project comes with built-in development server that is used to run application instantly without further configuration of other web servers. It means we don't need of apache or other web server to run application in development mode.

To run the application, we can use the following command.

1. $ python3 manage.py runserver



Server has started and can be accessed at localhost and port 8000. Let's access it using the browser, that looks like the below.



Application is running successfully. Now, we can customize it according to our requirement and can develop customized web application.

Virtual Environment Setup

It is recommended to setup virtual environment for creating Django application. This environment is used to create an isolated environment for Python/Django projects.

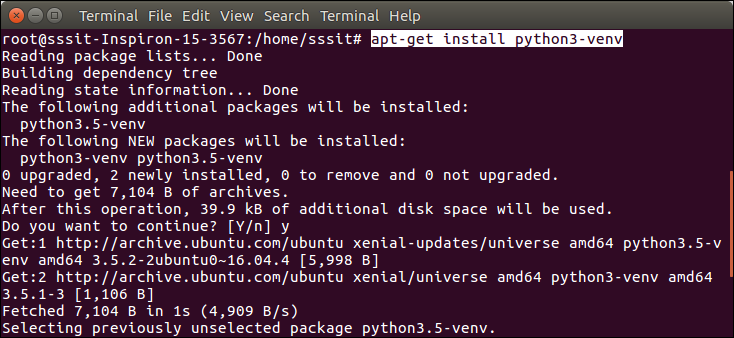
It separates dependencies of one project to other.

Following are the steps by steps process to setup virtual environment

**1. Install Package**

First, install **python3-venv** package to by using the following command.

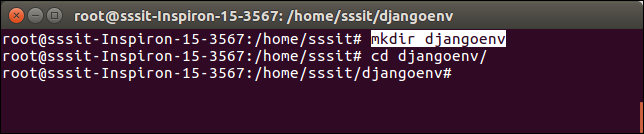
1. $ apt-get install python3-venv



**2. Create a Directory**

1. $ mkdir djangoenv

After it, change directory to the new created directory by using the **cd djangoenv.**



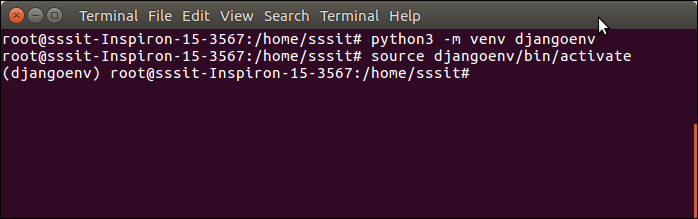
**3. Create Virtual Environment**

1. $ python3 -m venv djangoenv

**4. Activate Virtual Environment**

After creating virtual environment, activate it using the following command.

1. $ source djangoenv/bin/activate

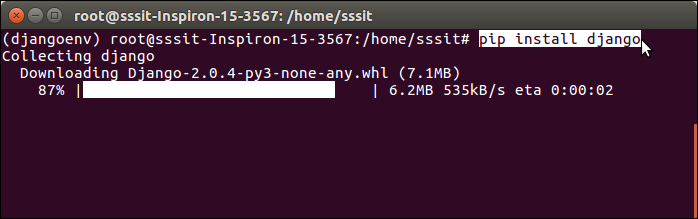


Well, virtual environment has started. Now, we can use it to create Django application.

Install Django

To install django, use the following command.

1. $ pip install django



and, here we go! Django has installed successfully. Now we can create new project and develop applications.

Django Admin Interface

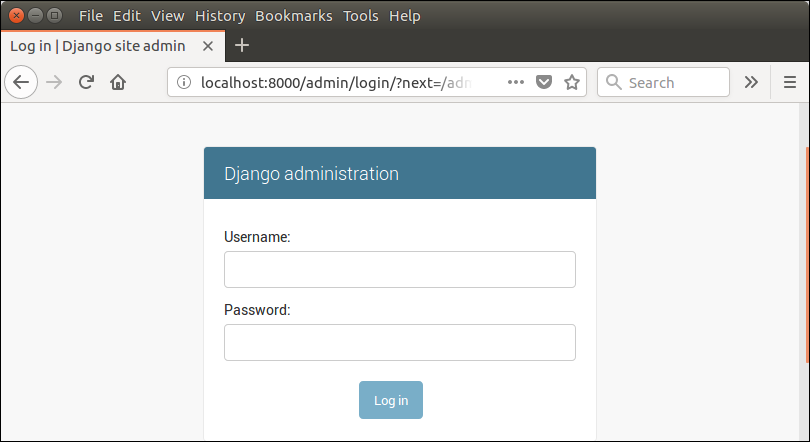
Django provides automatic admin interface that can be used to perform CRUD operations on the model. It reads metadata from the model to provide quick interface where user can manage content of the application.

This is built-in module and was design to perform admin related tasks to the user.

Lets see how to activate and use Django's admin interface.

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

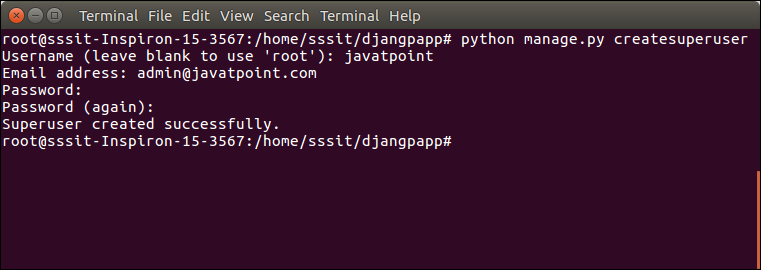
To access it at browser use '/**admin**/' at local machine like: **localhost:8000**/**admin**/ and it shows the following output:



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

Create an Admin User

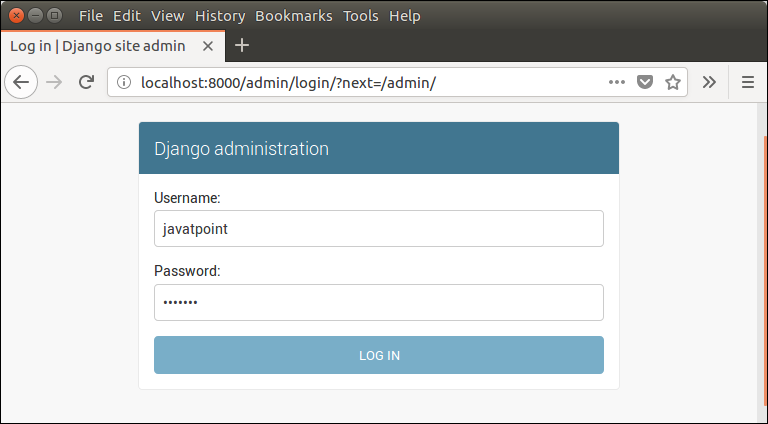
1. $ python3 managen.py createsuperuser



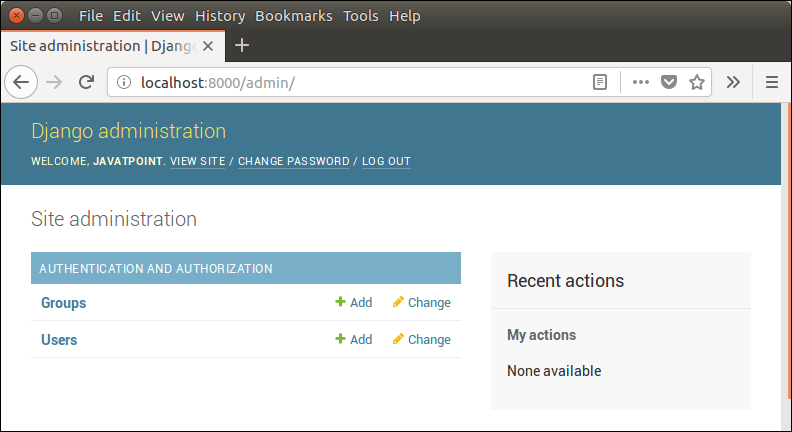
Now start development server and access admin login.

1. $ python3 manage.py runserver

Provide created username and password and login.



After login successfully, it shows the following interface.



It is a Django Admin Dashboard. Here, we can add and update the registered model. Model registration process will be discussed in further chapters.

Django App

In previous topics, we have seen the procedure to create Django project. Now, in this topic we will create app inside the created project.

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

The difference between a project and app is, project is a collection of configuration and apps whereas app is a Web application that is written to perform business logic.

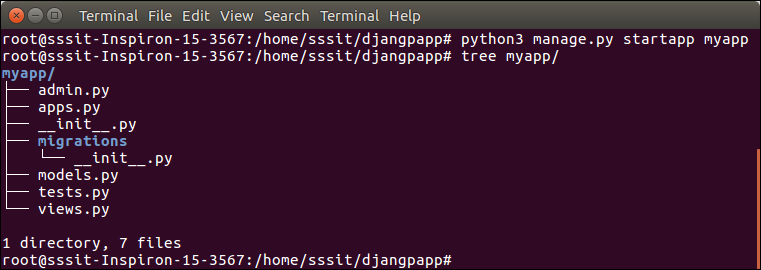
Creating an App

To create an app, use the following command.

1. $ python3 manage.py startapp appname

Django App Example

1. $ python3 manage.py startapp myapp



See the directory structure of created app, this contains the **migrations** folder to store migration files and model to write business logic.Initially, all the files are empty that can be used to implement business logic on the basis of MVC design pattern.To run this application, lets make some significant changes that displays **hello world** message on the browser.

Open **views.py** file in any editor and write the following code to it.

**// views.py**

from django.shortcuts **import** render

# Create your views here.

from django.http **import** HttpResponse

def hello(request):

**return** HttpResponse("<h2>Hello, Welcome to Django!</h2>")

**// urls.py**

from django.contrib **import** admin

from django.urls **import** path

from myapp **import** views

urlpatterns = [

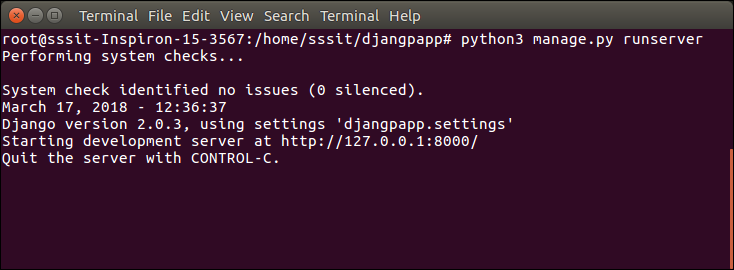
    path('admin/', admin.site.urls),

    path('hello/', views.hello),  ]

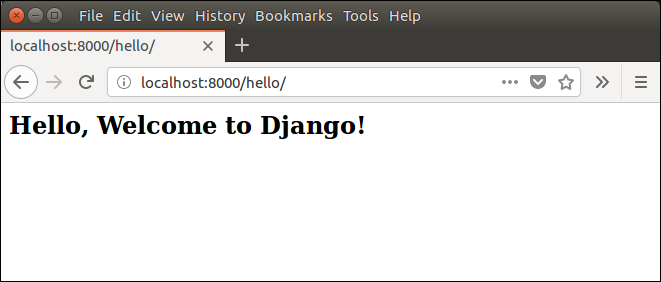
We have made changes in two files of our application. Now, lets run the application by using the following command.

Run the Application

1. $ python3 manage.py runserver



At browser enter the url **localhost:8000/hello** and it will show us the following output.



Django MVT

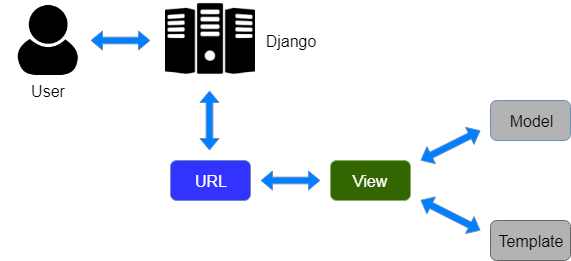
MVT (Model View Template) is a software design pattern which is a collection of three parts Model View and Template. The Model helps to handle database. It is a data access layer that handles the data.

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

Although Django follows MVC pattern but maintains it?s own conventions so controlling is handle by the framework itself.

There is no separate controller and complete application is based on Model View and Template. That?s why it is called MVT application.

See the following graph that shows the MVT based control flow.



Here, 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, **view is called** that interact with model and template, it renders a template.

Django respond back to the user and send a template as **response**.

Django Model

In Django, 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.

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

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

Lets see and example of Model. Here, we are creating Model **Employee** which has two fields **first\_name** and **last\_name**.

1. from django.db **import** models
3. **class** Employee(models.Model):
4. first\_name = models.CharField(max\_length=30)
5. last\_name = models.CharField(max\_length=30)

The **first\_name** and **last\_name** fields are specified as class attributes and each attribute maps to a database column.

This model will create a table into the database that looks like below.

1. CREATE TABLE appname\_employee (
2. "id" INT NOT NULL PRIMARY KEY,
3. "first\_name" varchar(30) NOT NULL,
4. "last\_name" varchar(30) NOT NULL
5. );

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

Register / Use Model

After creating model, register model into the **INSTALLED\_APPS** inside **settings.py.**

**For example,**

1. INSTALLED\_APPS = [
2. #...
3. 'appname',
4. #...
5. ]

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.

Django provides various built-in fields types.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Class** | **Particular** |
| AutoField | class AutoField(\*\*options) | It An IntegerField that automatically increments. |
| BigAutoField¶ | class BigAutoField(\*\*options) | It is a 64-bit integer, much like an  AutoField except that it is guaranteed to  fit numbers from 1 to 9223372036854775807. |
| BigIntegerField¶ | class BigIntegerField(\*\*options) | It is a 64-bit integer, much like an IntegerField  except that it is guaranteed to fit numbers  from -9223372036854775808 to  9223372036854775807. |
| BinaryField | class BinaryField(\*\*options) | A field to store raw binary data. |
| BooleanField | class BooleanField(\*\*options) | A true/false field. The default form widget for  this field is a CheckboxInput. |
| CharField | class DateField(auto\_now=False, auto\_now\_add=False, \*\*options) | It is a date, represented in Python by a  datetime.date instance. |
| DateTimeField | class DateTimeField(auto\_now=False, auto\_now\_add=False, \*\*options) | It is a date, represented in Python by a datetime.  date instance. |
| DateTimeField | class DateTimeField(auto\_now=False, auto\_now\_add=False, \*\*options) | It is used for date and time, represented  in Python by a datetime.datetime instance. |
| DecimalField¶ | class DecimalField(max\_digits=None, decimal\_places=None, \*\*options) | It is a fixed-precision decimal number,  represented in Python by a Decimal instance. |
| DurationField | class DurationField(\*\*options) | A field for storing periods of time. |
| EmailField¶ | class EmailField(max\_length=254, \*\*options) | It is a CharField that checks that the value is a  valid email address. |
| FileField¶ | class FileField(upload\_to=None, max\_length=100, \*\*options) | It is a file-upload field. |
| FloatField¶ | class FloatField(\*\*options) | It is a floating-point number represented in  Python by a float instance. |
| ImageField | class ImageField(upload\_to=None, height\_field=None, width\_field=None, max\_length=100, \*\*options) | It inherits all attributes and methods from  FileField, but also validates that the uploaded  object is a valid image. |
| IntegerField | class IntegerField(\*\*options) | It is an integer field. Values from -2147483648  to 2147483647 are safe in all databases  supported by Django. |
| NullBooleanField | class NullBooleanField(\*\*options) | Like a BooleanField, but allows NULL as one of  the options. |
| PositiveIntegerField | class PositiveIntegerField(\*\*options) | 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 | class SmallIntegerField(\*\*options) | It is like an IntegerField, but only allows  values under a certain (database-dependent)  point. |
| TextField | class TextField(\*\*options) | A large text field. The default form widget for this field is a Textarea. |
| TimeField | class TimeField(auto\_now=False, auto\_now\_add=False, \*\*options) | A time, represented in Python by a datetime.time  instance. |

Django Model Fields Example

1. first\_name = models.CharField(max\_length=50) # **for** creating varchar column
2. release\_date = models.DateField()                        # **for** creating date column
3. num\_stars = models.IntegerField()                       # **for** creating integer column

Field Options

Each field requires some argument that are used to set column attributes. For example, CharField requires mac\_length to specify varchar database.

Common arguments available to all field types. All are optional.

|  |  |
| --- | --- |
| **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. |

Django Model Example

We created a model Student that contains the following code in **models.py** file.

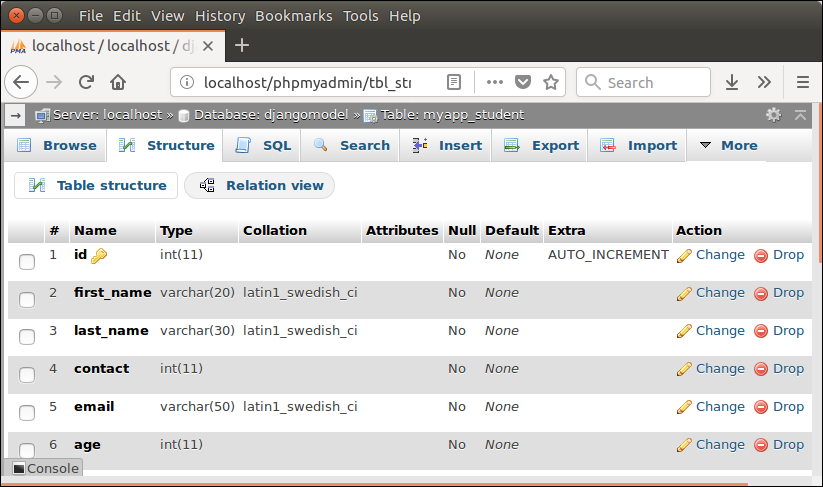
**//models.py**

1. **class** Student(models.Model):
2. first\_name = models.CharField(max\_length=20)
3. last\_name  = models.CharField(max\_length=30)
4. contact    = models.IntegerField()
5. email      = models.EmailField(max\_length=50)
6. age        = models.IntegerField()

After that apply migration by using the following command.

1. python3 manage.py makemigrations myapp

It will create a table **myapp\_student**. The table structure looks like the below.



Django Views

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

All the view function are created inside the **views.py** file of the django app.

Django View Simple Example

**//views.py**

1. **import** datetime
2. # Create your views here.
3. from django.http **import** HttpResponse
4. def index(request):
5. now = datetime.datetime.now()
6. html = "<html><body><h3>Now time is %s.</h3></body></html>" % now
7. **return** HttpResponse(html)    # rendering the template in HttpResponse

Let's step through the code.

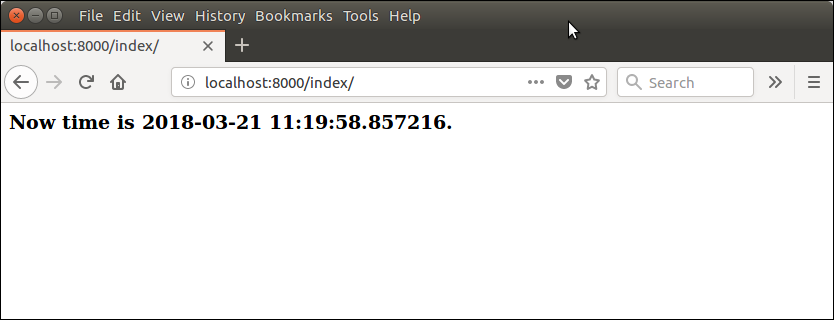
First, we import datatime library that provides method to get current date and time and HttpResponse class.

Next, we define a view function index that takes http request and respond back.

View calls, when gets mapped with url in **urls.py.** For example

1. path('index/', views.index),

**Output:**



Returning Errors

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

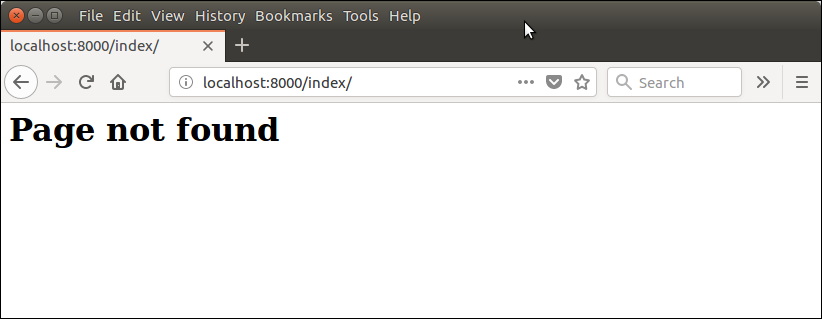
|  |  |
| --- | --- |
| **Class** | **Description** |
| class HttpResponseNotModified | It is used to designate that a page hasn't been modified since the user's last request (status code 304). |
| class HttpResponseBadRequest | It acts just like HttpResponse but uses a 400 status code. |
| class HttpResponseNotFound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

Django View Example

**// views.py**

1. from django.shortcuts **import** render
2. # Create your views here.
3. from django.http **import** HttpResponse, HttpResponseNotFound
4. def index(request):
5. a = 1
6. **if** a:
7. **return** HttpResponseNotFound('<h1>Page not found</h1>')
8. **else**:
9. **return** HttpResponse('<h1>Page was found</h1>') # rendering the template in HttpResponse

**Output:**



Django View HTTP Decorators

HTTP Decorators are used to restrict used access to view based on the request method.

These decorators are listed in django.views.decorators.http and return a django.http.HttpResponseNotAllowed if the conditions are not met.

**Syntax**

require\_http\_methods(request\_method\_list)

Django Http Decorator Example

**//views.py**

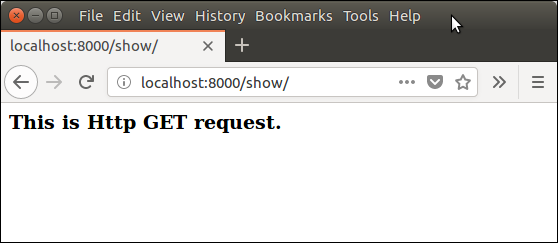
1. from django.shortcuts **import** render
2. # Create your views here.
3. from django.http **import** HttpResponse, HttpResponseNotFound
4. from django.views.decorators.http **import** require\_http\_methods
5. @require\_http\_methods(["GET"])
6. def show(request):
7. **return** HttpResponse('<h1>This is Http GET request.</h1>')

This method will execute only if request is a http GET request.

**//urls.py**

1. from django.contrib **import** admin
2. from django.urls **import** path
3. from myapp **import** views
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. path('index/', views.index),
7. path('show/',  views.show),
8. ]

**Output:**



Django Templates

Django provides a convenient way to generate dynamic HTML pages by using it's 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 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.

Django Template Configuration

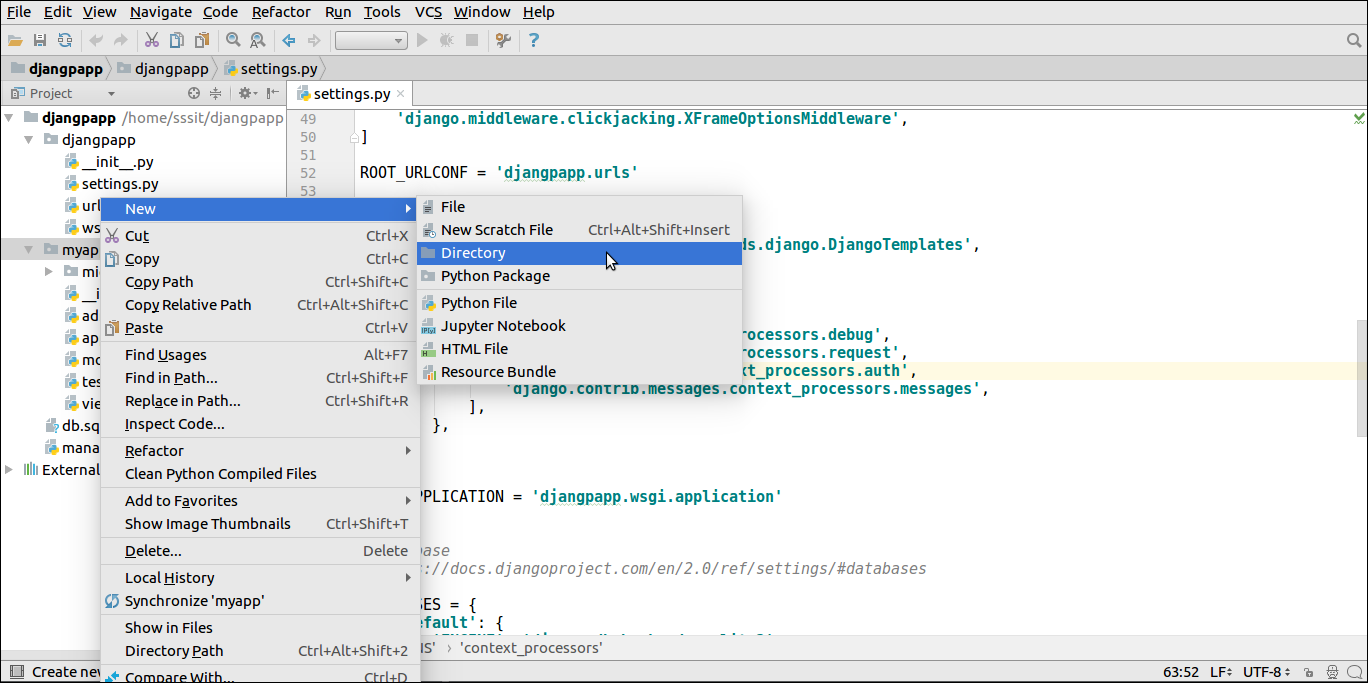
To configure template system, provides some entries in **settings.py** file.

1. TEMPLATES = [
2. {
3. 'BACKEND': 'django.template.backends.django.DjangoTemplates',
4. 'DIRS': [os.path.join(BASE\_DIR,'templates')],
5. 'APP\_DIRS': True,
6. 'OPTIONS': {
7. 'context\_processors': [
8. 'django.template.context\_processors.debug',
9. 'django.template.context\_processors.request',
10. 'django.contrib.auth.context\_processors.auth',
11. 'django.contrib.messages.context\_processors.messages',
12. ],
13. },
14. },
15. ]

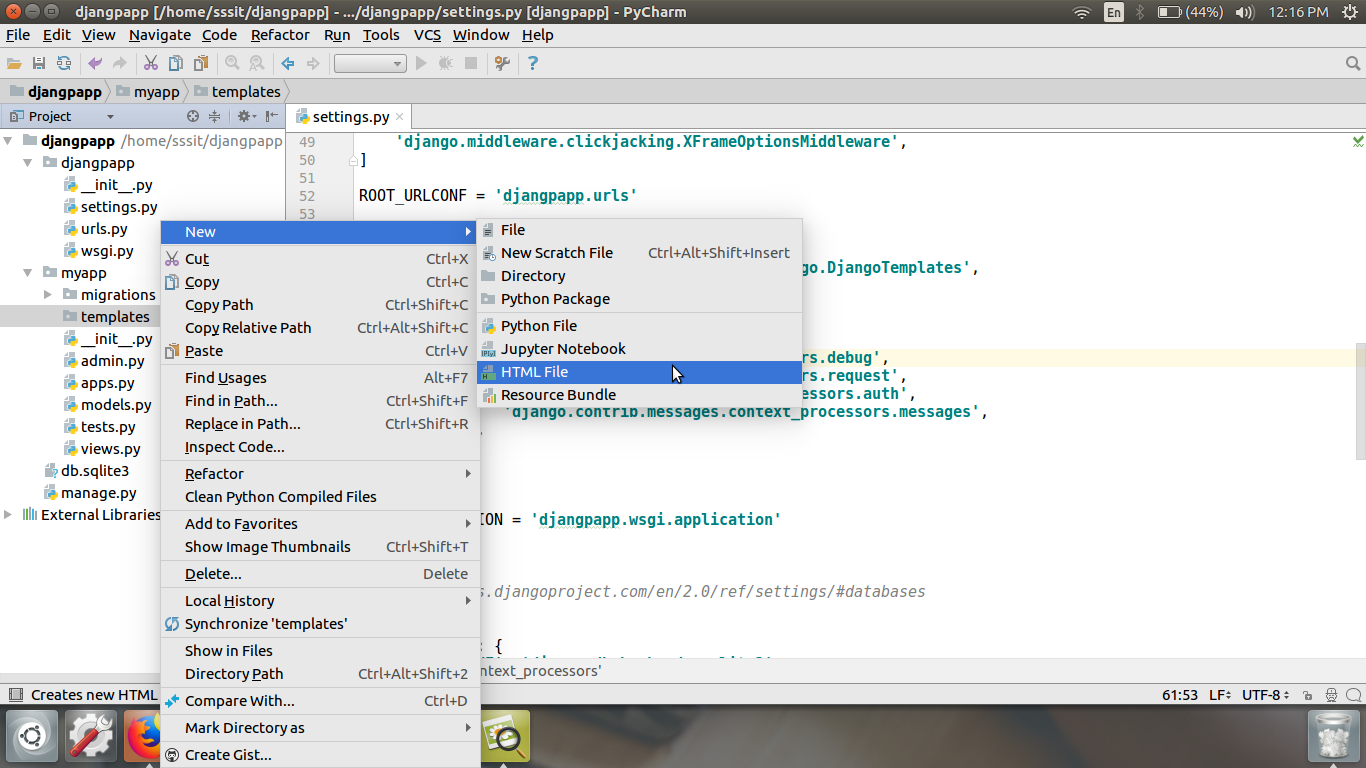
Here, we mentioned that our template directory name is **templates**. By default DjangoTemplates looks for a **templates** subdirectory in each of the INSTALLED\_APPS.

Django Template Simple Example

First create a directory **templates** inside the project app as we did below.



After that create a template **index.html** inside the created folder.



Our template **index.html** contains the following code.

**// index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. </head>
7. <body>
8. <h2>Welcome to Django!!!</h2>
9. </body>
10. </html>

Loading Template

To load template, call get\_template() method as we did below and pass template name.

**//views.py**

1. from django.shortcuts **import** render
2. #importing loading from django template
3. from django.template **import** loader
4. # Create your views here.
5. from django.http **import** HttpResponse
6. def index(request):
7. template = loader.get\_template('index.html') # getting our template
8. **return** HttpResponse(template.render())       # rendering the template in HttpResponse

Set a url to access the template from the browser.

**//urls.py**

1. path('index/', views.index),

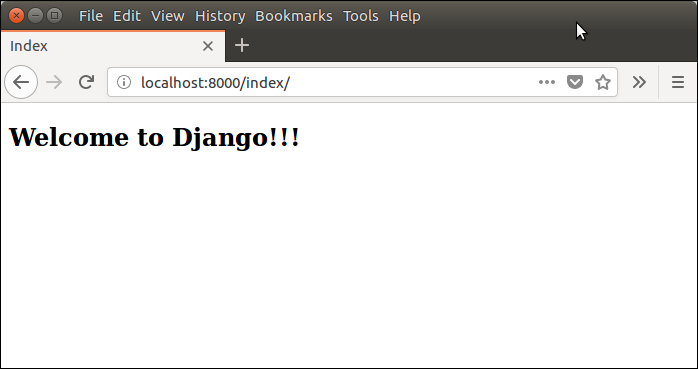
Register app inside the INSTALLED\_APPS

1. INSTALLED\_APPS = [
2. 'django.contrib.admin',
3. 'django.contrib.auth',
4. 'django.contrib.contenttypes',
5. 'django.contrib.sessions',
6. 'django.contrib.messages',
7. 'django.contrib.staticfiles',
8. 'myapp'
9. ]

Run Server

Execute the following command and access the template by entering **localhost:8000/index** at the browser.

1. $ python3 manage.py runserver



Django Template Language

Django template uses it's own syntax to deal with variable, tags, expressions etc. Template is rendered with a context that is used to get value at web page. See the examples.

Variables

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

1. My name is {{name}}.
2. My name is rahul

Django Variable Example

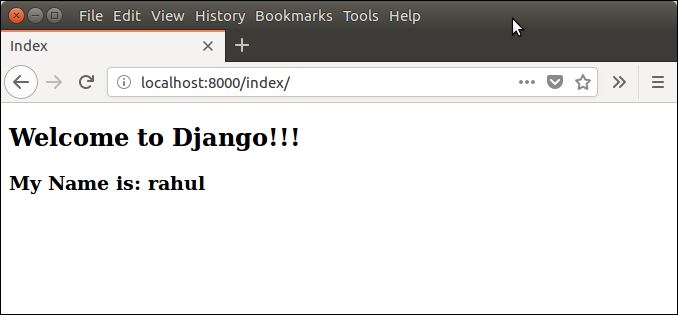
**//views.py**

1. from django.shortcuts **import** render
2. #importing loading from django template
3. from django.template **import** loader
4. # Create your views here.
5. from django.http **import** HttpResponse
6. def index(request):
7. template = loader.get\_template('index.html') # getting our template
8. name = {
9. 'student':'rahul'
10. }
11. **return** HttpResponse(template.render(name))       # rendering the template in HttpResponse

**//index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. </head>
7. <body>
8. <h2>Welcome to Django!!!</h2>
9. <h3>My Name is: {{ student }}</h3>
10. </body>
11. </html>

**Output:**



Tags

In 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. For example.

1. {% csrf\_token %}
3. {% **if** user.is\_authenticated %}
4. Hello, {{ user.username }}.
5. {% endif %}

Django URL Mapping

Well, till here, we have learnt to create model, view and template. Now, we will learn about routing of application.

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

Lets open the file **urls.py** of the project and see the what it looks like:

**// urls.py**

1. from django.contrib **import** admin
2. from django.urls **import** path
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. ]

See, Django already has mentioned a URL here for the admin. The path function takes first argument as route of string or regex type.

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

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

Django URL Functions

Here, we are giving some commonly used functions for URL handling and mapping.

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Example** |
| path(route, view, kwargs=None, name=None) | It returns an element for inclusion in urlpatterns. | path('index/', views.index,  name='main-view') |
| re\_path(route, view, kwargs=None, name=None) | It returns an element for inclusion in urlpatterns. | re\_path(r'^index/$', views.index,  name='index'), |
| include(module, namespace=None) | It is a function that takes a full Python import path to another URLconf module that should be "included" in this place. |  |
| register\_converter(converter, type\_name) | It is used for registering a converter for use in path() routes. |  |

Django Static Files Handling

In web application, apart from business logic and data handling, we also need to handle and manage static resources like: CSS, JavaScript, images etc in our application.

It is important to manage these resources so that it does not effect our application performance.

Django deals this very efficiently and provides a convenient manner to use resources.

The **django.contrib.staticfiles** module helps to manage them.

Django Static (CSS, JavaScript, images) Configuration

1. Include the **django.contrib.staticfiles** in **INSTALLED\_APPS.**

1. INSTALLED\_APPS = [
2. 'django.contrib.admin',
3. 'django.contrib.auth',
4. 'django.contrib.contenttypes',
5. 'django.contrib.sessions',
6. 'django.contrib.messages',
7. 'django.contrib.staticfiles',
8. 'myapp'
9. ]

2. Define STATIC\_URL in settings.py file. For example.

1. STATIC\_URL = '/static/'

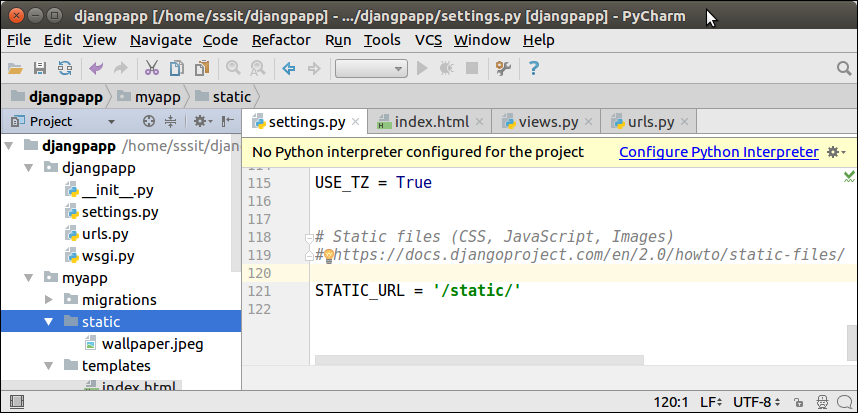
3. Load static files in templates by using the following code.

1. {% load **static** %}

4. Store all images, JavaScript, CSS files in **static** folder of the application. First create a directory **static,** store the files inside it.



Our project structure looks like this.



Django Image Loading Example

To, load image in template file, use the following code. For example.

**// index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. {% load **static** %}
7. </head>
8. <body>
9. <img src="{% static '/wallpaper.jpeg' %}" alt="My image" height="300px" width="700px"/>
10. </body>
11. </html>

**//urls.py**

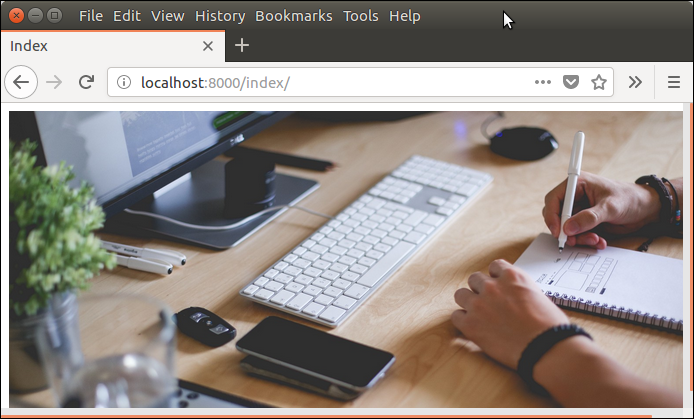
1. from django.contrib **import** admin
2. from django.urls **import** path
3. from myapp **import** views
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. path('index/', views.index),
7. ]

**//views.py**

1. def index(request):
2. **return** render(request,'index.html')

Run the server by using **python manage.py runserver** command.

After that access the template by **localhost:8000/index** URL, and it will produce the following output to the browser.



Django Loading JavaScript

To load JavaScript file, just add the following line of code in **index.html** file.

1. {% load **static** %}
2. <script src="{% static '/js/script.js' %}"

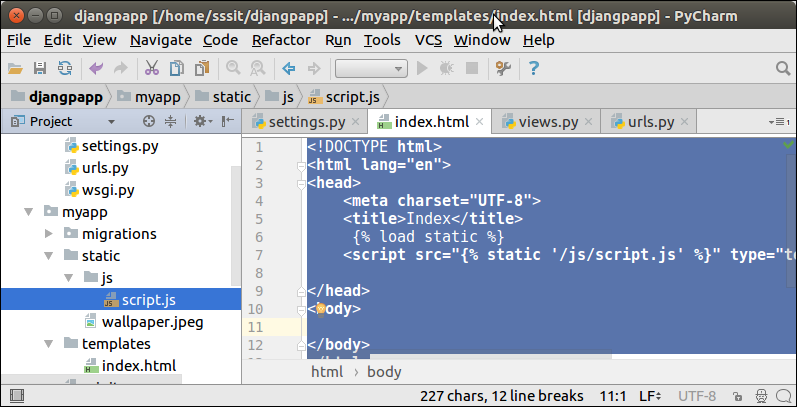
**// index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. {% load **static** %}
7. <script src="{% static '/js/script.js' %}" type="text/javascript"></script>
8. </head>
9. <body>
10. </body>
11. </html>

**// script.js**

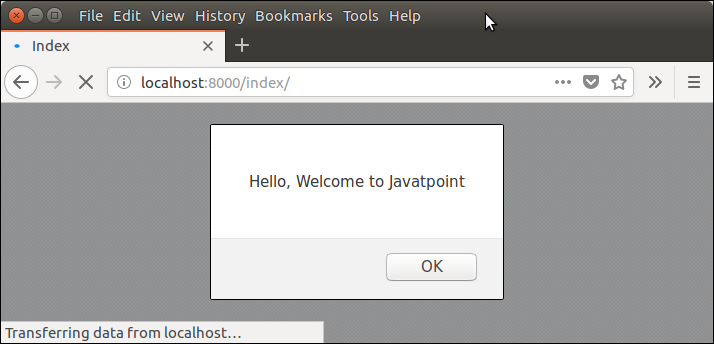
1. alert("Hello, Welcome to Javatpoint");

Now, our project structure looks like the this:



Run the server by using **python manage.py runserver** command.

After that access the template by **localhost:8000/index** URL, and it will produce the following output to the browser.



Django Loading CSS Example

To, load CSS file, use the following code in **index.html** file.

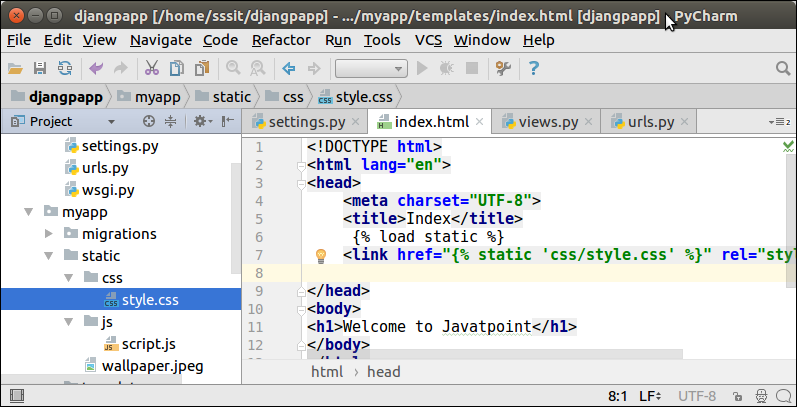
1. {% load **static** %}
2. <link href="{% static 'css/style.css' %}" rel="stylesheet">

After the create a directory css and file style.css that contains the following code.

**// style.css**

1. h1{
2. color:red;
3. }

Our project structure looks like this:

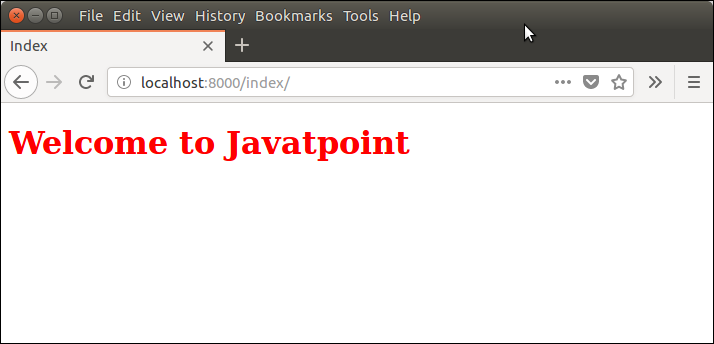


**// index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. {% load **static** %}
7. <link href="{% static 'css/style.css' %}" rel="stylesheet">
8. </head>
9. <body>
10. <h1>Welcome to Javatpoint</h1>
11. </body>
12. </html>

Run the server by using **python manage.py runserver** command.

After that access the template by **localhost:8000/index** URL, and it will produce the following output to the browser.



Django Model Form

It is a class which is used to create HTML form by using Model. It is a art 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 form file.

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

let's see an example.

Django ModelForm Example

First create a model that contains fields name and other metadata. It can be used to create table in database and dynamic html form.

**// model.py**

1. from \_\_future\_\_ **import** unicode\_literals
2. from django.db **import** models
4. **class** Student(models.Model):
5. first\_name = models.CharField(max\_length=20)
6. last\_name  = models.CharField(max\_length=30)
7. **class** Meta:
8. db\_table = "student"

This file contains a class that inherits ModelForm and mention the model name for which html for is created.

**// form.py**

1. from django **import** forms
2. from myapp.models **import** Student
4. **class** EmpForm(forms.ModelForm):
5. **class** Meta:
6. model = Student
7. fields = "\_\_all\_\_"

Write a view function to load the ModelForm from forms.py.

**//views.py**

1. from django.shortcuts **import** render
2. from myapp.form **import** StuForm
4. def index(request):
5. stu = StuForm()
6. **return** render(request,"index.html",{'form':stu})

**//urls.py**

1. from django.contrib **import** admin
2. from django.urls **import** path
3. from myapp **import** views
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. path('index/', views.index),
7. ]

And finally, create a **index.html** file that contains the following code.

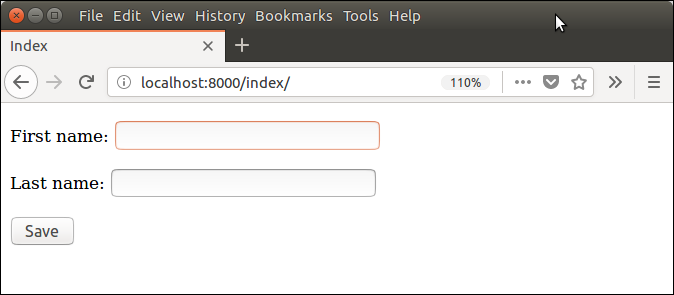
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. </head>
7. <body>
8. <form method="POST" **class**="post-form">
9. {% csrf\_token %}
10. {{ form.as\_p }}
11. <button type="submit" **class**="save btn btn-default">Save</button>
12. </form>
13. </body>
14. </html>

Run Server

Run the server by using **python manage.py runserver** command.

After that access the template by **localhost:8000/index** URL, and it will produce the following output to the browser.

**Output:**



Well, an HTML for is created automatically. This is the smartness of Django.

# Django Forms-160

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 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 perform validation after submitting the form.

Lets see an example, in which we are creating some fields too.

1. from django **import** forms
2. **class** StudentForm(forms.Form):
3. firstname = forms.CharField(label="Enter first name",max\_length=50)
4. lastname  = forms.CharField(label="Enter last name", max\_length = 100)

A StudentForm is created that contains two fields of CharField type. Charfield is a class and used to create HTML text input component in the form.

The label is used to set HTML label of the component and max\_length sets length of input value.

When rendered, it produces the following HTML to the browser.

1. <label **for**="id\_firstname">Enter first name:</label>
2. <input type="text" name="firstname" required maxlength="50" id="id\_firstname" />
3. <label **for**="id\_lastname">Enter last name:</label> <input type="text" name="lastname" required maxlength="100" id="id\_lastname" />

#### Note: Django Form does not include <form> tags, or a submit button. We'll have to provide those ourselves in the template.

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Class** | **HTML Input** | **Empty value** |
| BooleanField | class BooleanField(\*\*kwargs) | CheckboxInput | False |
| CharField | class CharField(\*\*kwargs) | TextInput | Whatever you've given as empty\_value. |
| ChoiceField | class ChoiceField(\*\*kwargs) | Select | '' (an empty string) |
| DateField | class DateField(\*\*kwargs) | DateInput | None |
| DateTimeField | class DateTimeField(\*\*kwargs) | DateTimeInput | None |
| DecimalField | class DecimalField(\*\*kwargs) | NumberInput | None |
| EmailField | class EmailField(\*\*kwargs) | EmailInput | '' (an empty string) |
| FileField | class FileField(\*\*kwargs) | ClearableFileInput | None |
| ImageField | class ImageField(\*\*kwargs) | ClearableFileInput | None |

Lets see a complete example to create an HTML form with the help of django Form class.

## Building a Form in Django

Suppose we want to create a form to get Student information, use the following code.

1. from django **import** forms
2. **class** StudentForm(forms.Form):
3. firstname = forms.CharField(label="Enter first name",max\_length=50)
4. lastname  = forms.CharField(label="Enter last name", max\_length = 100)

Put this code into a **forms.py** file.

## Instantiating Form in Django

Now, we need to instantiate the form in **views.py** file. See, the below code.

**// views.py**

1. from django.shortcuts **import** render
2. from myapp.form **import** StudentForm
4. def index(request):
5. student = StudentForm()
6. **return** render(request,"index.html",{'form':student})

Passing the context of form into index template that looks like this:

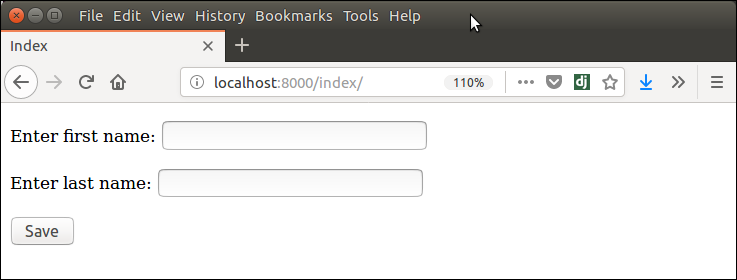
**// index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. </head>
7. <body>
8. <form method="POST" **class**="post-form">
9. {% csrf\_token %}
10. {{ form.as\_p }}
11. <button type="submit" **class**="save btn btn-default">Save</button>
12. </form>
13. </body>
14. </html>

Provide the URL in urls.py

1. from django.contrib **import** admin
2. from django.urls **import** path
3. from myapp **import** views
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. path('index/', views.index),
7. ]

Run Server and access the form at browser by **localhost:8000/index,** and it will produce the following output.



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

#### Note: that you'll have to provide the surrounding <table> or <ul> elements yourself.

Django Form Validation

Django provides built-in methods to validate form data automatically. At first, django forms submit only if it contains CSRF tokens and after that it uses clean and easy approach to validate data.

The **is\_valid()** method is used to perform validation for each field of the form, it is define in Django Form class. It returns True if data is valid and place all data into cleaned\_data attribute.

Lets see an example that takes user input and validate as well.

Django Validation Example

This example contains the following files and code. First create a Model.

**// models.py**

1. from django.db **import** models
2. **class** Employee(models.Model):
3. eid = models.CharField(max\_length=20)
4. ename = models.CharField(max\_length=100)
5. econtact = models.CharField(max\_length=15)
6. **class** Meta:
7. db\_table = "employee"

After that create a form.

**// forms.py**

1. from django **import** forms
2. from myapp.models **import** Employee
4. **class** EmployeeForm(forms.ModelForm):
5. **class** Meta:
6. model = Employee
7. fields = "\_\_all\_\_"

Instantiate the form

Instantiate the form and validate method of form request and data as well

**//views.py**

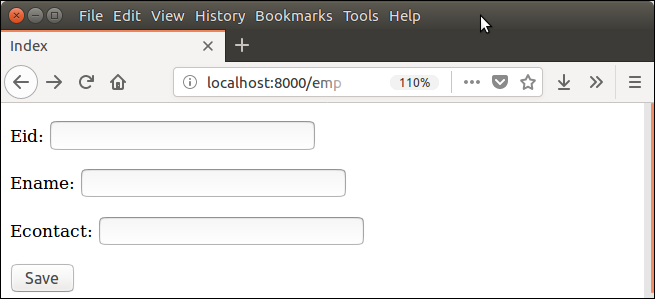
1. def emp(request):
2. **if** request.method == "POST":
3. form = EmployeeForm(request.POST)
4. **if** form.is\_valid():
5. **try**:
6. **return** redirect('/')
7. except:
8. pass
9. **else**:
10. form = EmployeeForm()
11. **return** render(request,'index.html',{'form':form})

Index template that shows form and errors.

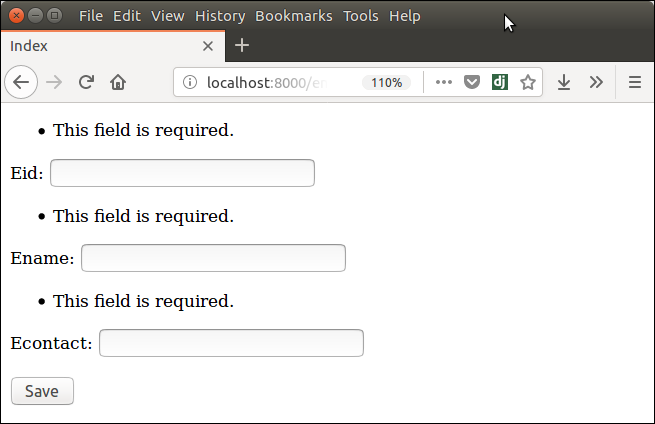
**// index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. </head>
7. <body>
8. <form method="POST" **class**="post-form" enctype="multipart/form-data">
9. {% csrf\_token %}
10. {{ form.as\_p }}
11. <button type="submit" **class**="save btn btn-default">Save</button>
12. </form>
13. </body>
14. </html>

Start server and access the form.



It validates each field and throws errors.



Django File Upload

File upload to the server using Django is pretty easy task. Django provides built-in library and methods that help to allow a user to upload file to the server.

The **forms.FileField()** is used to create file input which is used to submit file to the server.

While working with files, make sure the form tag contains **enctype="multipart/form-data"**property.

Lets see an example of uploading file to the server. This example contains the following files.

**Template (index.html)**

1. <body>
2. <form method="POST" **class**="post-form" enctype="multipart/form-data">
3. {% csrf\_token %}
4. {{ form.as\_p }}
5. <button type="submit" **class**="save btn btn-default">Save</button>
6. </form>
7. </body>

**Form (forms.py)**

1. from django **import** forms
2. **class** StudentForm(forms.Form):
3. firstname = forms.CharField(label="Enter first name",max\_length=50)
4. lastname  = forms.CharField(label="Enter last name", max\_length = 10)
5. email     = forms.EmailField(label="Enter Email")
6. file      = forms.FileField() # **for** creating file input

**View (views.py)**

Pass one extra argument **request.FILES** to the form constructor. This argument contains the uploaded file instance.

1. from django.shortcuts **import** render
2. from django.http **import** HttpResponse
3. from myapp.functions.functions **import** handle\_uploaded\_file
4. from myapp.form **import** StudentForm
5. def index(request):
6. **if** request.method == 'POST':
7. student = StudentForm(request.POST, request.FILES)
8. **if** student.is\_valid():
9. handle\_uploaded\_file(request.FILES['file'])
10. **return** HttpResponse("File uploaded successfuly")
11. **else**:
12. student = StudentForm()
13. **return** render(request,"index.html",{'form':student})

**Specify URL (urls.py)**

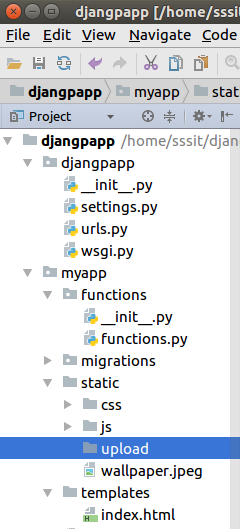
1. from django.contrib **import** admin
2. from django.urls **import** path
3. from myapp **import** views
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. path('index/', views.index),
7. ]

**Upload Script (functions.py)**

This function is used to read the uploaded file and store at provided location. Put this code into the **functions.py** file. But first create this file into the project.

1. def handle\_uploaded\_file(f):
2. with open('myapp/static/upload/'+f.name, 'wb+') as destination:
3. **for** chunk in f.chunks():
4. destination.write(chunk)

Create a directory **upload** to store uploaded file. Our project looks like this:

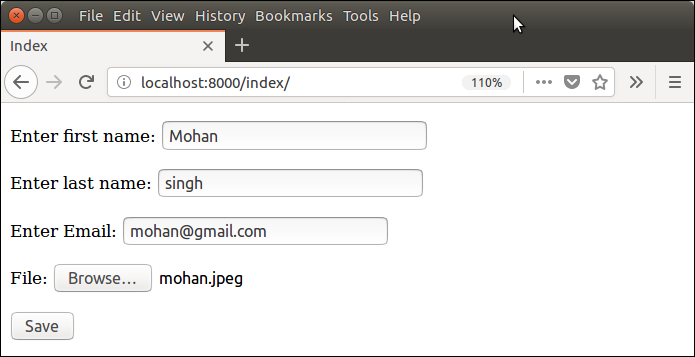


Initially, this directory is empty lets upload a file to it.

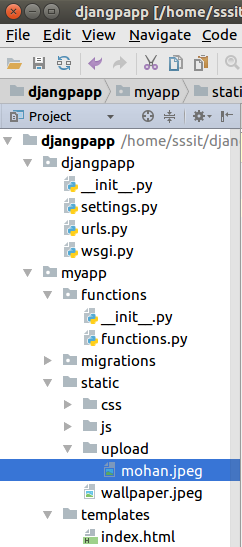
**Start Server**

1. python manage.py runserver

**Output**



Submit this form and see the **upload** folder. Now, it contains the uploaded file.



# Django Database Connectivity

The **settings.py** file contains all the project settings along with database connection details. By default Django works with **Sqlite,** but also allows to configure for other databases as well.

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

In case of MySQL, **django.db.backends.mysql** driver is used to establish connection between application and database. Lets see an example.

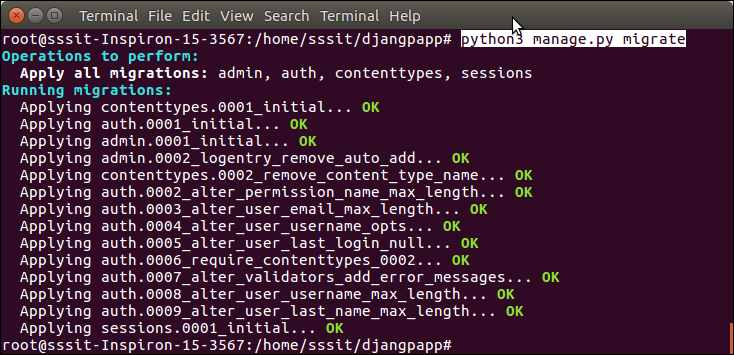
We will connect our application to MySQL, so have provided connection details. The settings.py file of our project contains the following code for the database.

1. DATABASES = {
2. 'default': {
3. 'ENGINE': 'django.db.backends.mysql',
4. 'NAME': 'djangoApp',
5. 'USER':'root',
6. 'PASSWORD':'mysql',
7. 'HOST':'localhost',
8. 'PORT':'3306'
9. }
10. }

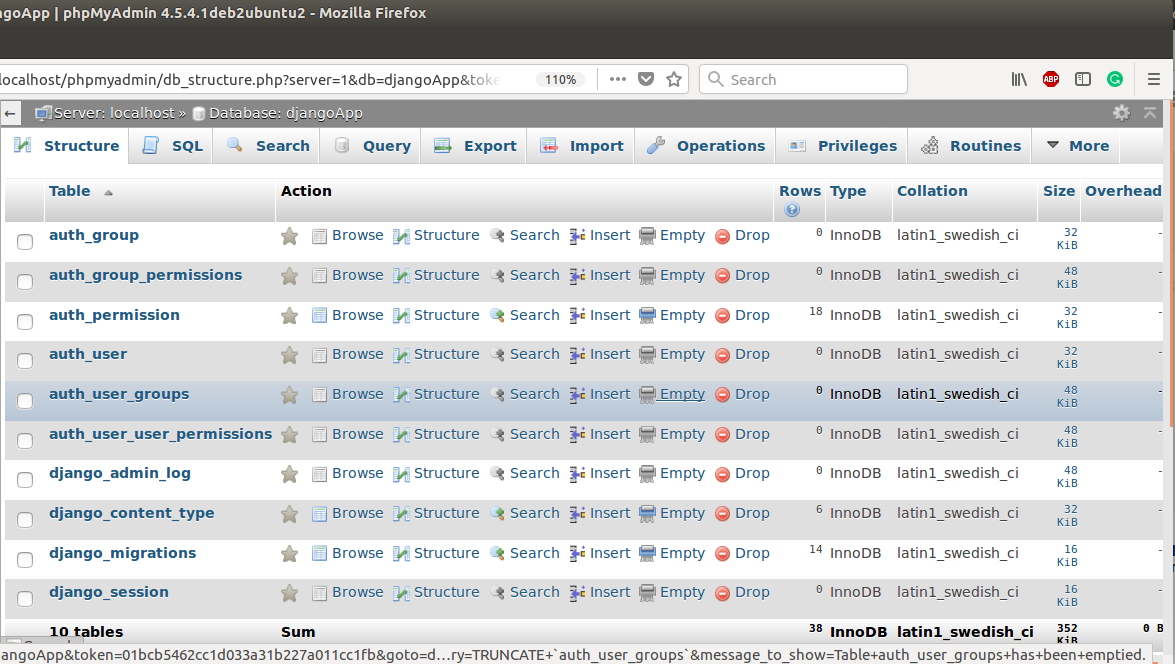
After providing details, check the connection using migrate command.

1. $ python3 manage.py migrate

This command will create tables for admin, auth, contenttypes, and sessions. See the example.



Access to the MySQL database and see the database. Now it contains the following tables.



#### ****Note:**** It throws error, if database connectivity fails: django.db.utils.OperationalError: (1045, "Access denied for user 'root'@'localhost' (using password: YES)")

## Migrating Model

Well, till here, we have learnt to connect Django application to the MySQL database. Next, we will see how to create table using model.

Each Django's model is mapped to the a table in the database. So after creating a model, we need to migrate it. Lets see an example.

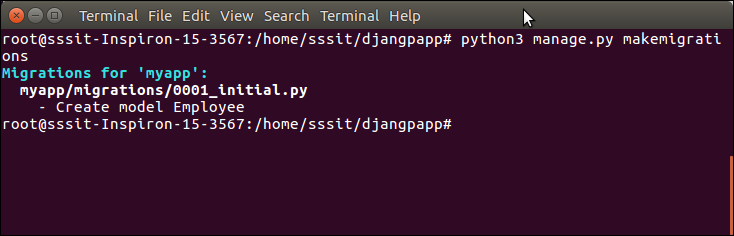
Suppose, we have a model class Employee in the **models.py** file that contains the following code.

**// models.py**

1. from django.db **import** models
2. **class** Employee(models.Model):
3. eid      = models.CharField(max\_length=20)
4. ename    = models.CharField(max\_length=100)
5. econtact = models.CharField(max\_length=15)
6. **class** Meta:
7. db\_table = "employee"

Django first creates a migration file that contains the details of table structure. To create migration use the following command.

1. $ python3 manage.py makemigrations



The created migration file is located into **migrations** folder and contains the following code.

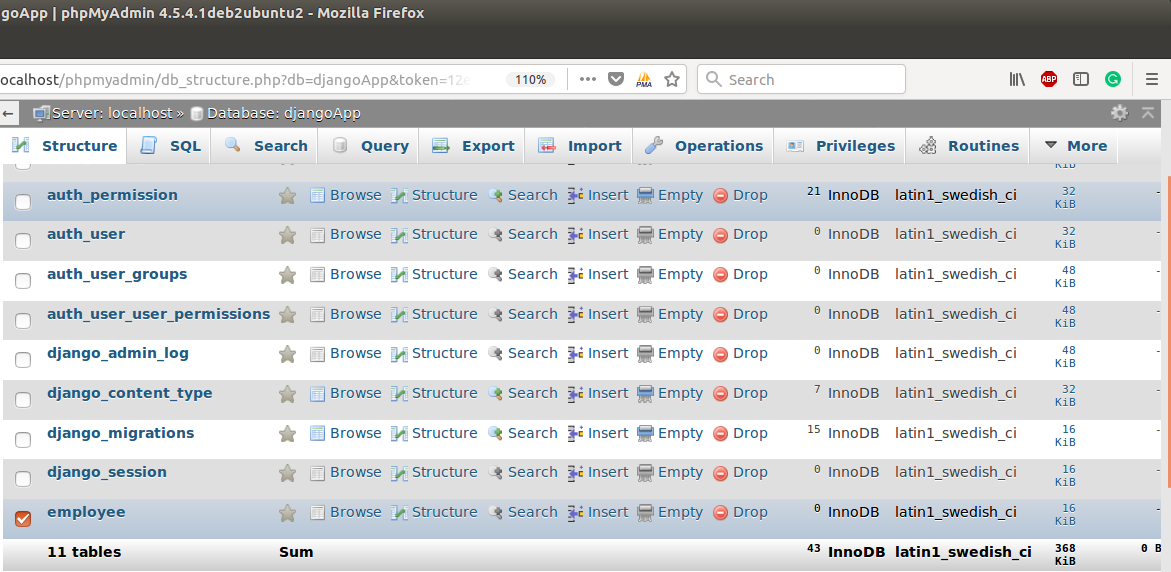
1. from django.db **import** migrations, models
2. **class** Migration(migrations.Migration):
3. initial = True
4. dependencies = [
5. ]
6. operations = [
7. migrations.CreateModel(
8. name='Employee',
9. fields=[
10. ('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),
11. ('eid', models.CharField(max\_length=20)),
12. ('ename', models.CharField(max\_length=100)),
13. ('econtact', models.CharField(max\_length=15)),
14. ],
15. options={
16. 'db\_table': 'employee',
17. },
18. ),
19. ]

Now, migrate to reflect the changes into the database.

1. $ python3 manage.py migrate



Check the database again, now it contains the **employee** table.



See, table is present. Well, we have successfully established connection between our application and MySQL database.

Django Database Migrations

Migration is the way of applying changes that we made to our 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 various command that are used to perform migration related tasks. After creating model, we can use these commands.

* **makemigrations :** It is used to create migration file that contains code for tabled schema of a model.
* **migrate :** It creates table according to the schema defined in migration file.
* **sqlmigrate :** It is used to show raw sql query of the applied migration.
* **showmigrations :** It list out all the migrations and their status.

Suppose, we have a model that contains the following attributes.

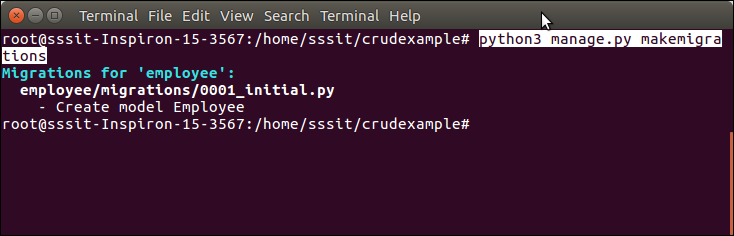
Model

**//models.py**

1. from django.db **import** models
2. **class** Employee(models.Model):
3. eid = models.CharField(max\_length=20)
4. ename = models.CharField(max\_length=100)
5. econtact = models.CharField(max\_length=15)
6. **class** Meta:
7. db\_table = "employee"

To create migration for this model, use the following command. It will create a migration file inside the migration folder.

1. $ python3 manage.py makemigrations



This migration file contains the code in which a Migration class is created that contains the name and fields of employee table.

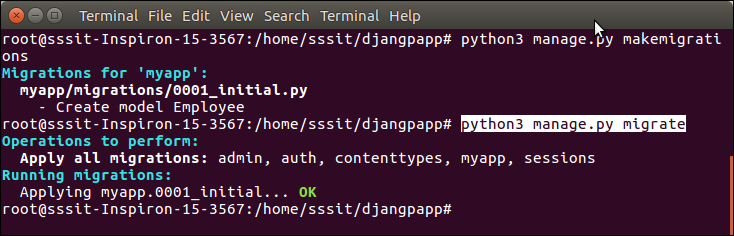
Migrations

***// 0001\_initial.py***

1. from django.db **import** migrations, models
2. **class** Migration(migrations.Migration):
3. initial = True
4. dependencies = [
5. ]
6. operations = [
7. migrations.CreateModel(
8. name='Employee',
9. fields=[
10. ('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),
11. ('eid', models.CharField(max\_length=20)),
12. ('ename', models.CharField(max\_length=100)),
13. ('econtact', models.CharField(max\_length=15)),
14. ],
15. options={
16. 'db\_table': 'employee',
17. },
18. ),
19. ]

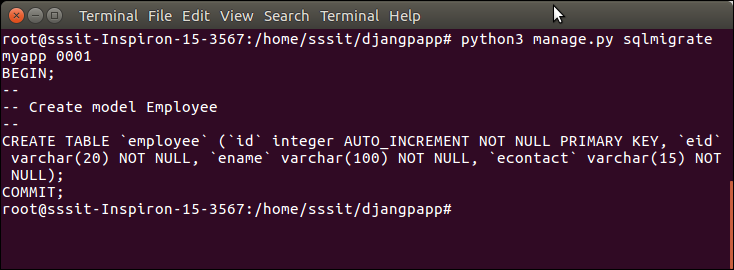
After creating migration, migrate it so that it reflects to the database permanently. The migrate command is given below.

1. $ python3 manage.py migrate



Apart from creating migration, we can see raw sql query executing behind the applied migration. The **sqlmigrate app-name migration-name** is used to get raw sql query. See an example.

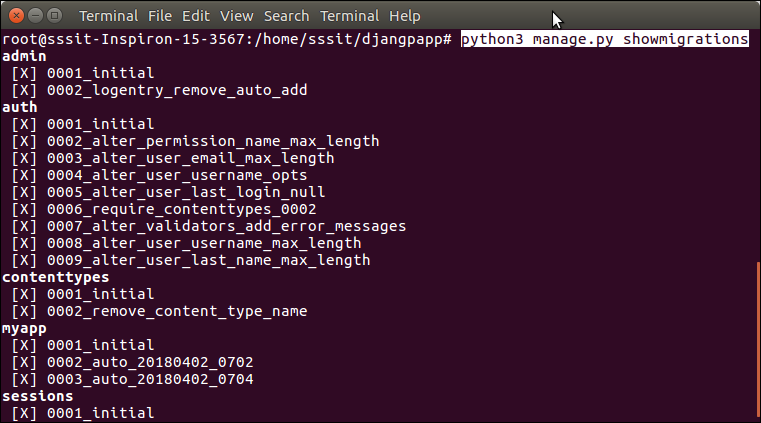
1. $ python3 manage.py migrate



And showmigrations command is used to show applied migrations. See the example.

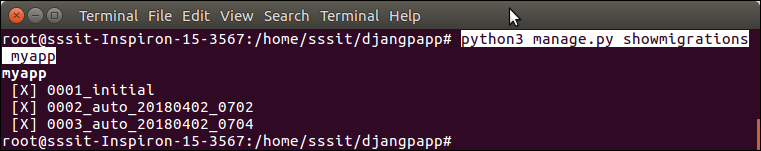
If no app-name is provided, it shows all migrations applied on the project.

1. $ python3 manage.py showmigrations



We can get app specific migrations by specifying app-name, see the example.

1. $ python3 manage.py showmigrations myapp



Django Middleware

in Django, middleware is a light weight plugin that process during request and response execution. Middleware is used to perform function in the application, functions may include security, session, csrf protection, authentication etc.

Django provides various built-in middleware and also allows us to write our own middleware. See, **settings.py** file of Django project that contains various middleware, that are used to provides functionalities to the application. For example, Security Middleware is used to maintain security of the application.

**// settings.py**

1. MIDDLEWARE = [
2. 'django.middleware.security.SecurityMiddleware',
3. 'django.contrib.sessions.middleware.SessionMiddleware',
4. 'django.middleware.common.CommonMiddleware',
5. 'django.middleware.csrf.CsrfViewMiddleware',
6. 'django.contrib.auth.middleware.AuthenticationMiddleware',
7. 'django.contrib.messages.middleware.MessageMiddleware',
8. 'django.middleware.clickjacking.XFrameOptionsMiddleware',
9. ]

Creating Own Middleware

middleware is a class that takes an argument **get\_response** and returns a response.

1. **class** FirstMiddleware:
2. def \_\_init\_\_(self, get\_response):
3. self.get\_response = get\_response
5. def \_\_call\_\_(self, request):
6. response = self.get\_response(request)
7. **return** response

**\_\_init\_\_(get\_response)**

It must accept the get\_response argument because django initializes middleware with only it. It calls only once whereas \_\_call\_\_ executes for each request.

Activating Middleware

To activate middleware, add it to the MIDDLEWARE list of the **settings.py** file.

1. MIDDLEWARE = [
2. 'django.middleware.security.SecurityMiddleware',
3. 'django.contrib.sessions.middleware.SessionMiddleware',
4. 'django.middleware.common.CommonMiddleware',
5. 'django.middleware.csrf.CsrfViewMiddleware',
6. 'django.contrib.auth.middleware.AuthenticationMiddleware',
7. 'django.contrib.messages.middleware.MessageMiddleware',
8. 'django.middleware.clickjacking.XframeOptionsMiddleware',
9. 'add new created middleware here'
10. ]

A Django project does not require middleware, the MIDDLEWARE list can be empty, but recommended that have at least a CommonMiddleware.

Middleware Order and Layering

Middleware applies in the order it is defined in MIDDLEWARE list and each middleware class is a layer. The MIDDLEWARE list is like an onion so each request pass through from top to bottom and response is in reverse order (bottom to up).

Other Middleware Methods

Apart from request and response, we can add three more methods to add more features in our middleware.

**process\_view(request, view\_func, view\_args, view\_kwargs )**

It takes, HttpRequest object, function object, list of arguments passed to view, dictionary of arguments respectively.

This method executes just before the calling of view. It returns either None or HttpResponse, if it returns an HttpResponse, it stops processing and return the result.

**process\_template\_response(request,response)**

It takes two arguments first is reference of HttpRequest and second is HttpResponse object. This method is called just after the view finished execution.

It returns a response object which implements the render method.

**process\_exception(request, exception)**

This method takes two arguments, first is HttpRequest object and second is Exception class object that is raised by the view function.

This method returns either None or HttpResponse object. If it returns response, the middleware will be applied and result return to the browser. Otherwise exception is handle by default handling system.

Django Request and Response

Request and response both phases are part of client server architecture that is followed by Django web application.

When a client requests for a resource, a HttpRequest object is created and correspond view function is called that returns HttpResponse object.

To handle request and response, Django provides HttpRequest and HttpResponse classes. Each class has it?s own attributes and methods.

Lets have a look at the HttpRequest class.

Django HttpRequest

This class is defined in the **django.http** module and used to handle client request. Following are the attributes of this class.

Django HttpRequest Attributes

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| HttpRequest.scheme | A string representing the scheme of the request (http or https usually). |
| HttpRequest.body | It returns the raw HTTP request body as a byte string. |
| HttpRequest.path | It returns the full path to the requested page, does not include the  scheme or domain. |
| HttpRequest.path\_info | It shows path info portion of the path. |
| HttpRequest.method | It shows the HTTP method used in the request. |
| HttpRequest.encoding | It shows the current encoding used to decode form submission data. |
| HttpRequest.content\_type | It shows the MIME type of the request, parsed from the CONTENT\_TYPE  header. |
| HttpRequest.content\_params | It returns a dictionary of key/value parameters included in the CONTENT\_TYPE  header. |
| HttpRequest.GET | It returns a dictionary-like object containing all given HTTP GET parameters. |
| HttpRequest.POST | It is a dictionary-like object containing all given HTTP POST parameters. |
| HttpRequest.COOKIES | It returns all cookies available. |
| HttpRequest.FILES | It contains all uploaded files. |
| HttpRequest.META | It shows all available Http headers. |
| HttpRequest.resolver\_match | It contains an instance of ResolverMatch representing the resolved URL. |

And following table contains the methods of HttpRequest class.

Django HttpRequest Methods

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| HttpRequest.get\_host() | It returns the original host of the request. |
| HttpRequest.get\_port() | It returns the originating port of the request. |
| HttpRequest.get\_full\_path() | It returns the path, plus an appended query string, if applicable. |
| HttpRequest.build\_absolute\_uri *(location)* | It returns the absolute URI form of location. |
| HttpRequest.get\_signed\_cookie *(key, default=RAISE\_ERROR,salt='', max\_age=None)* | It returns a cookie value for a signed cookie, or raises  a django.core.signing.BadSignature exception if the signature  is no longer valid. |
| HttpRequest.is\_secure() | It returns True if the request is secure; that is, if it was made with HTTPS. |
| HttpRequest.is\_ajax() | It returns True if the request was made via an XMLHttpRequest. |

Django HttpRequest Example

**// views.py**

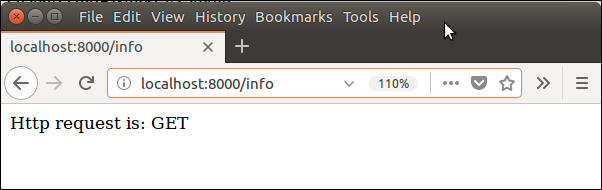
1. def methodinfo(request):
2. **return** HttpResponse("Http request is: "+request.method)

**// urls.py**

1. path('info',views.methodinfo)

Start the server and get access to the browser. It shows the request method name at the browser.

**Output:**



Django HttpResponse

This class is a part of **django.http** module. It is responsible for generating response correspond to the request and back to the client.

This class contains various attributes and methods that are given below.

Django HttpResponse Attributes

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| HttpResponse.content | A bytestring representing the content, encoded from a string if necessary. |
| HttpResponse.charset | It is a string denoting the charset in which the response will be encoded. |
| HttpResponse.status\_code | It is an **HTTP status code** for the response. |
| HttpResponse.reason\_phrase | The HTTP reason phrase for the response. |
| HttpResponse.streaming | It is false by default. |
| HttpResponse.closed | It is True if the response has been closed. |

Django HttpResponse Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| HttpResponse.\_\_init\_\_*(content='', content\_type=None, status=200, reason=None, charset=None)* | It is used to instantiate an HttpResponse object with the  given page content and content type. |
| HttpResponse.\_\_setitem\_\_*(header, value)* | It is used to set the given header name to the given value. |
| HttpResponse.\_\_delitem\_\_*(header)* | It deletes the header with the given name. |
| HttpResponse.\_\_getitem\_\_*(header)* | It returns the value for the given header name. |
| HttpResponse.has\_header*(header)* | It returns either True or False based on a case-insensitive  check for a header with the provided name. |
| HttpResponse.setdefault*(header, value)* | It is used to set default header. |
| HttpResponse.write*(content)* | It is used to create response object of file like object. |
| HttpResponse.flush() | It is used to flush the response object. |
| HttpResponse.tell() | This method makes an HttpResponse instance a file-like object. |
| HttpResponse.getvalue() | It is used to get the value of HttpResponse.content. |
| HttpResponse.readable() | This method is used to create stream-like object of HttpResponse  class. |
| HttpResponse.seekable() | It is used to make response object seekable. |

We can use these methods and attributes to handle response in Django application.

Django Exceptions

Exception is a abnormal event that leads to program failure. To deal with this situation, Django uses its own exception classes and supports all core Python exceptions as well.

Django core exceptions classes are defined in **django.core.exceptions** module. This module contains the following classes.

Django Exception Classes

|  |  |
| --- | --- |
| **Exception** | **Description** |
| AppRegistryNotReady | It is raised when attempting to use models before the app loading process. |
| ObjectDoesNotExist | The base class for DoesNotExist exceptions. |
| EmptyResultSet | If a query does not return any result, this exception is raised. |
| FieldDoesNotExist | It raise when the requested field does not exist. |
| MultipleObjectsReturned | This exception is raised by a query if only one object is expected,  but multiple objects are returned. |
| SuspiciousOperation | This exception is raised when a user has performed an operation that  should be considered suspicious from a security perspective. |
| PermissionDenied | It is raised when a user does not have permission to perform the action requested. |
| ViewDoesNotExist | It is raised by django.urls when a requested view does not exist. |
| MiddlewareNotUsed | It is raised when a middleware is not used in the server configuration. |
| ImproperlyConfigured | The ImproperlyConfigured exception is raised when Django is somehow  improperly configured. |
| FieldError | It is raised when there is a problem with a model field. |
| ValidationError | It is raised when data validation fails form or model field validation. |

Django URL Resolver Exceptions

These exceptions are defined in **django.urls** module.

|  |  |
| --- | --- |
| **Exception** | **Description** |
| Resolver404 | This exception raised when the path passed to resolve() function does not map to a view. |
| NoReverseMatch | It is raised when a matching URL in your URLconf cannot be identified based  on the parameters supplied. |

Django Database Exceptions

The following exceptions are defined in **django.db** module.

|  |  |
| --- | --- |
| **Exception** | **Description** |
| DatabaseError | It occurs when database is not available. |
| IntegrityError | It occurs when a insertion query executes. |
| DataError | It raises when data related issues comes in database. |

Django Http Exceptions

The following exceptions is defined in **django.http** module.

|  |  |
| --- | --- |
| **Exception** | **Description** |
| UnreadablePostError | It is raised when a user cancels an upload. |

Django Transaction Exceptions

The transaction exceptions are defined in django.db.transaction.

|  |  |
| --- | --- |
| **Exception** | **Description** |
| TransactionManagementError | It is raised for any and all problems related to  database transactions. |

Django Exception Example

Suppose, we want to get employee record where id = 12, our view function will look below. It raises DoesNotExist exception if data not found. This is Django's built-in exception.

**// views.py**

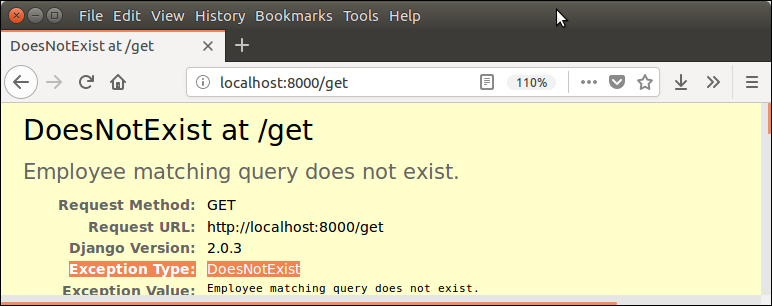
1. def getdata(request):
2. data = Employee.objects.get(id=12)
3. **return** HttpResponse(data)

**// urls.py**

1. path('get',views.getdata)

It shows the following exception because no record is available at id 12.

**Output:**

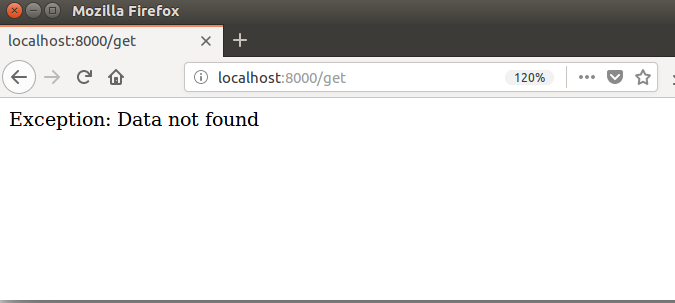


We can handle it by using try and except, now lets handle this exception.

**// Views.py**

1. def getdata(request):
2. **try**:
3. data = Employee.objects.get(id=12)
4. except ObjectDoesNotExist:
5. **return** HttpResponse("Exception: Data not found")
6. **return** HttpResponse(data);

**Output:**



Django Session

Session is a mechanism to store information at server side during the interaction with web application.

In Django, by default session stores in database and also allows file based and cache based sessions. It is implemented via a piece of middleware and can be enabled by using the following code.

Put **django.contrib.sessions.middleware.SessionMiddleware** in MIDDLEWARE and **django.contrib.sessions** in INSTALLED\_APPS of settings.py file.

To set and get session in views, we can use **request.session** and can set multiple times too.

The **class backends.base.SessionBase** is a base class of all session objects. It contains following standard methods.

|  |  |
| --- | --- |
| **Method** | **Description** |
| \_\_getitem\_\_(key) | It is used to get session value. |
| \_\_setitem\_\_(key, value) | It is used to set session value. |
| \_\_delitem\_\_(key) | It is used to delete session object. |
| \_\_contains\_\_(key) | It check whether container contains the particular session object or not. |
| get(key, default=None) | It is used to get session value of specified key. |

Lets see an example in which we will set and get session values. Two functions are defined in **views.py** file.

Django Session Example

First is used to set and second is used to get session values

**//views.py**

1. from django.shortcuts **import** render
2. from django.http **import** HttpResponse
4. def setsession(request):
5. request.session['sname'] = 'irfan'
6. request.session['semail'] = 'irfan.sssit@gmail.com'
7. **return** HttpResponse("session is set")
8. def getsession(request):
9. studentname = request.session['sname']
10. studentemail = request.session['semail']
11. **return** HttpResponse(studentname+" "+studentemail);

Url mapping to call both the functions.

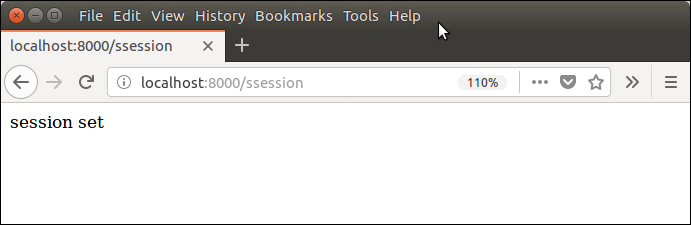
**// urls.py**

1. from django.contrib **import** admin
2. from django.urls **import** path
3. from myapp **import** views
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. path('index/', views.index),
7. path('ssession',views.setsession),
8. path('gsession',views.getsession)
9. ]

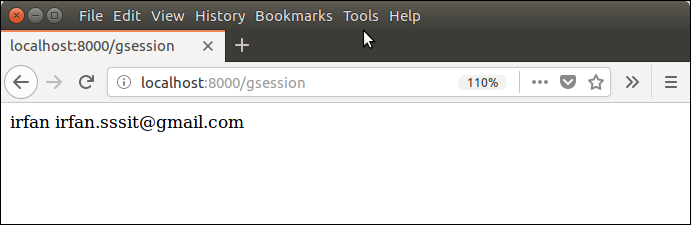
**Run Server**

1. $ python3 manage.py runserver

and set session by using **localhost:8000/ssession**



Session has been set, to check it, use **localhost:8000/gsession**



Django Cookie

Cookie is a small piece of information which is stored at client browser. It is used to store user's data in file permanently (or for specified time).

Cookie has it's expiry date and time and remove automatically when gets expire.

Django provides built-in methods to set and fetch cookie.

The **set\_cookie()** method is used to set cookie and **get()** method is used to get cookie.

The **request.COOKIES['key']** array can also be used to get cookie values.

Django Cookie Example

In **views.py,** two functions setcookie() and getcookie() are used to set and get cookie respectively

**// views.py**

1. from django.shortcuts **import** render
2. from django.http **import** HttpResponse
4. def setcookie(request):
5. response = HttpResponse("Cookie Set")
6. response.set\_cookie('java-tutorial', 'javatpoint.com')
7. **return** response
8. def getcookie(request):
9. tutorial  = request.COOKIES['java-tutorial']
10. **return** HttpResponse("java tutorials @: "+  tutorial);

And urls specified to access these functions.

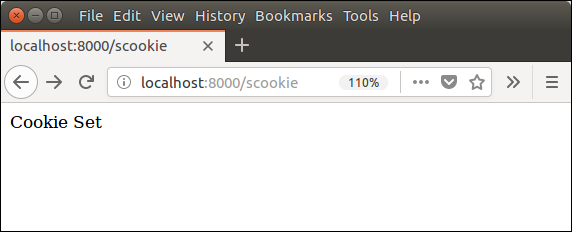
**// urls.py**

1. from django.contrib **import** admin
2. from django.urls **import** path
3. from myapp **import** views
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. path('index/', views.index),
7. path('scookie',views.setcookie),
8. path('gcookie',views.getcookie)
9. ]

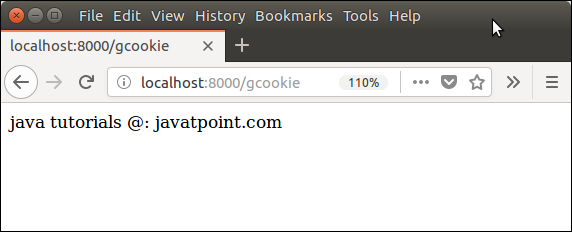
**Start Server**

1. $ python3 manage.py runserver

After starting server, set cookie by using **localhost:8000/scookie** url. It shows the following output to the browser.



And get cookie by using **localhost:8000/gcookie** url. It shows the set cookie to the browser.



Create CSV with Django

Django uses python's built-in csv library to create Dynamic CSV (Comma Separated values) file. We can use this library into our project's view file.

Lets see an example, here we have a django project to that we are implementing this feature. A view function **getfile()** is created.

Django CSV Example

In this example, we are creating CSV using static data.

**// Views.py**

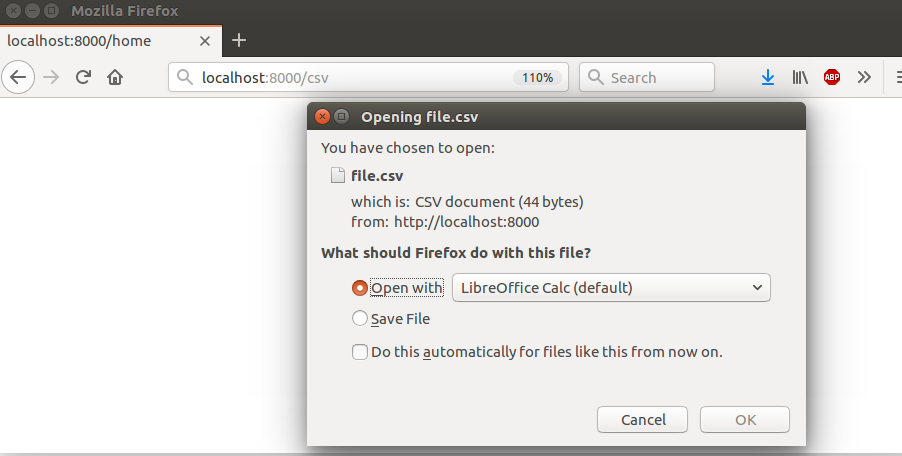
1. **import** csv
3. def getfile(request):
4. response = HttpResponse(content\_type='text/csv')
5. response['Content-Disposition'] = 'attachment; filename="file.csv"'
6. writer = csv.writer(response)
7. writer.writerow(['1001', 'John', 'Domil', 'CA'])
8. writer.writerow(['1002', 'Amit', 'Mukharji', 'LA', '"Testing"'])
9. **return** response

**// urls.py**

Provide url for the function.

1. path('csv',views.getfile)

While executing to the browser, it renders a CSV file. See the example.



Apart from static data, we can get CSV from database too. See, the following example in which we are getting data from the table by using the **Employee** model.

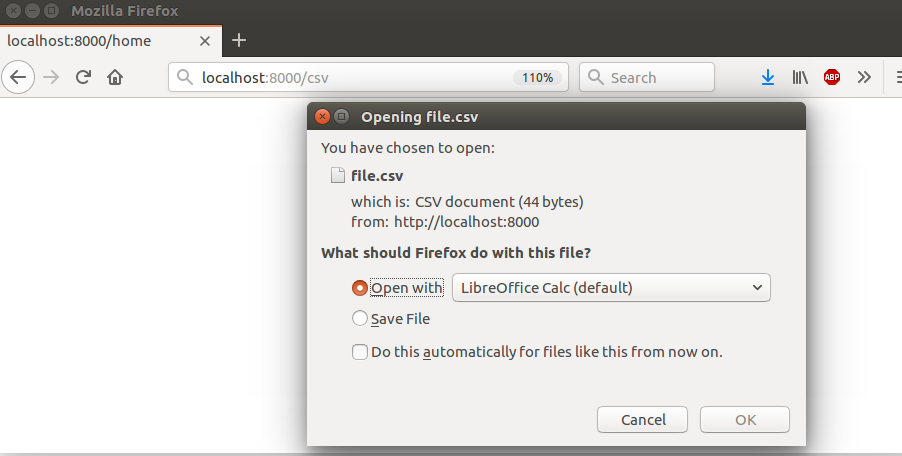
Dynamic CSV using Database

**// views.py**

**from**myapp.models **import**Employee **import**csv

1. def getfile(request):
2. response = HttpResponse(content\_type='text/csv')
3. response['Content-Disposition'] = 'attachment; filename="file.csv"'
4. employees = Employee.objects.all()
5. writer = csv.writer(response)
6. **for** employee in employees:
7. writer.writerow([employee.eid,employee.ename,employee.econtact])
8. **return** response

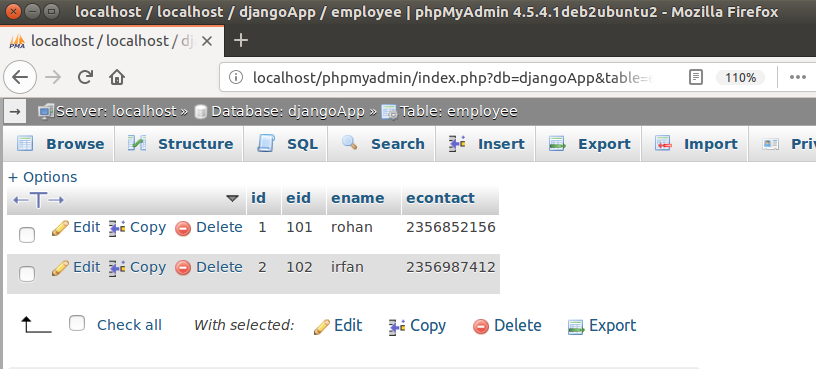
**Output:**



Save the file and open into the text editor that contains the following data.



This data is retrieve from the table employee, a snapshot of table is shown below.



Well, we have seen that this library is very useful to create dynamic CSV file. Now, implement it into django project when required.

Django PDF

Here, we will learn how to design and generate PDF file using Django view. To generate PDF, we will use **ReportLab** Python PDF library that creates customized dynamic PDF.

It is an open source library and can be downloaded easily by using the following command in Ubuntu.

1. $ pip install reportlab

After installing, we can import it by import keyword in the view file.

Below is a simple PDF example, in which we are outputting a string message "Hello form javatpoint". This library provides a canvas and tools that are used to generate customize PDF. See the example.

**// views.py**

1. from reportlab.pdfgen **import** canvas
2. from django.http **import** HttpResponse
4. def getpdf(request):
5. response = HttpResponse(content\_type='application/pdf')
6. response['Content-Disposition'] = 'attachment; filename="file.pdf"'
7. p = canvas.Canvas(response)
8. p.setFont("Times-Roman", 55)
9. p.drawString(100,700, "Hello, Javatpoint.")
10. p.showPage()
11. p.save()
12. **return** response

First provide MIME (content) type as application/pdf, so that output generates as PDF rather that HTML,

Set Content-Disposition in which provide header as attachment and output file name.

Pass response argument to the canvas and drawstring to write the string after that apply save method and return response.

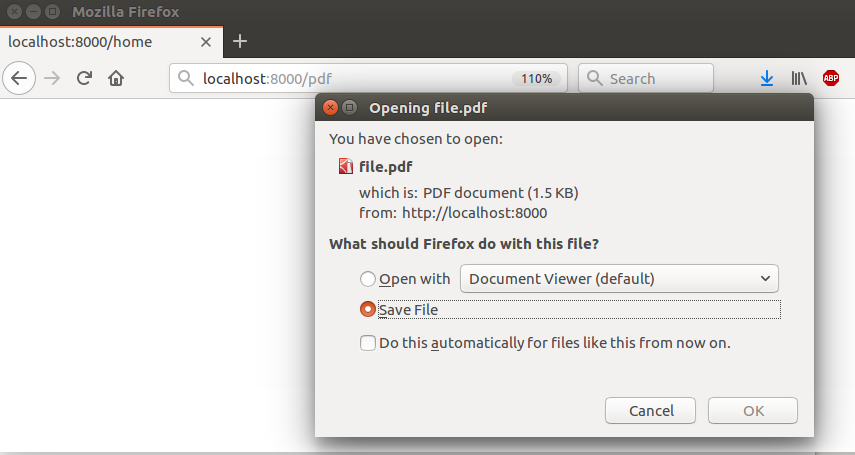
**// urls.py**

1. path('pdf',views.getpdf)

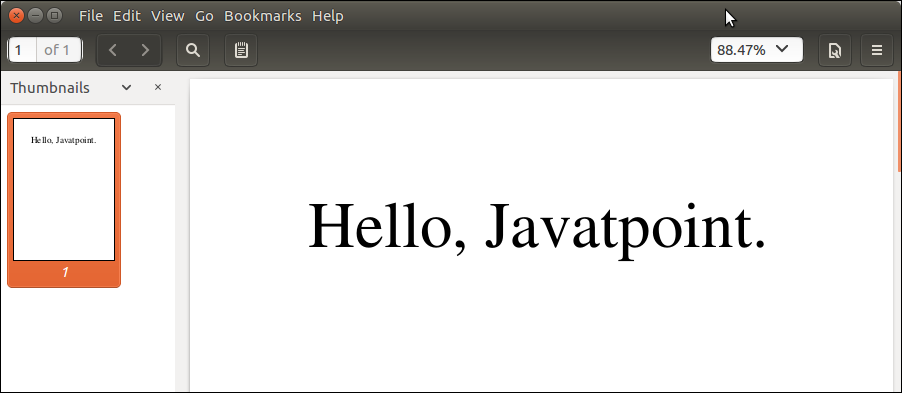
Set the above code in urls.py to call view function.

Run server and access this view on the browser that creates a pdf file. See the examples.

**Output:**



A PDF file is generated and ready to download. Download the file and open it, it shows the string message that we wrote.



Apart from it, this library contains the lots of other methods to design and generate PDF dynamically.

Django with Bootstrap

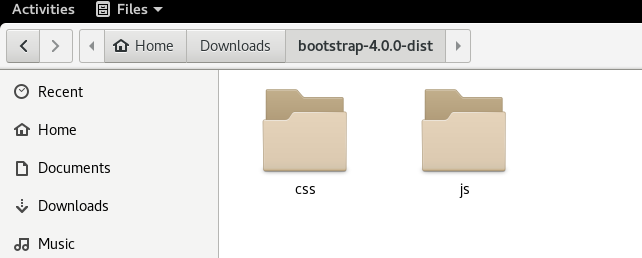
Bootstrap is a framework which is used to create user interface in web applications. It provides css, js and other tools that help to create required interface.

In Django, we can use bootstrap to create more user friendly applications.

To implement bootstrap, we need to follow the following steps.

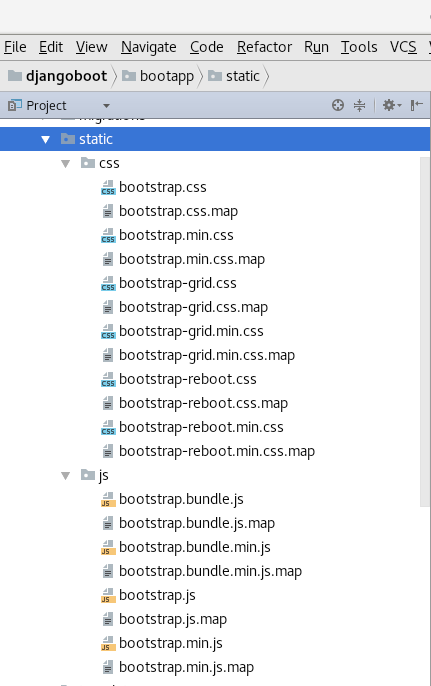
**1. Download the Bootstrap**

Visit the official site https://getbootstrap.com to download the bootstrap at local machine. It is a zip file, extract it and see it contains the two folder.



**2. Create a Directory**

Create a directory with the name **static** inside the created app and place the css and jss folders inside it. These folders contain numerous files, see the screen shot.



**3. Create a Template**

First create a templates folder inside the app then create a index.htm file to implement (link) the bootstrap css and js files.

**4. Load the Boostrap**

load the bootstrap files resides into the static folder. Use the following code.

1. {% load staticfiles %}

And link the files by providing the file location (source). See the index.html file.

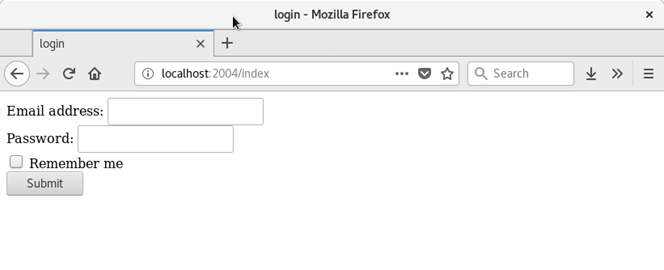
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Title</title>
6. {% load staticfiles %}
7. <link href="{% static 'css/bootstrap.min.css' %}" >
8. <script src="{% static 'bootstrap.min.js' %}"></script>
9. <script>alert();</script>
10. </head>
11. <body>
12. </body>
13. </html>

In this template, we have link two files one is bootstrap.min.css and second is bootstrap.min.js. Lets see how to use them in application.

Suppose, if we don't use bootstrap, our html login for looks like this:

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>login</title>
6. </head>
7. <body>
8. <form action="/save" method="post">
9. <div **class**="form-group">
10. <label **for**="email">Email address:</label>
11. <input type="email" **class**="form-control" id="email">
12. </div>
13. <div **class**="form-group">
14. <label **for**="pwd">Password:</label>
15. <input type="password" **class**="form-control" id="pwd">
16. </div>
17. <div **class**="checkbox">
18. <label><input type="checkbox"> Remember me</label>
19. </div>
20. <button type="submit" **class**="btn btn-primary">Submit</button>
21. </form>
22. </body>
23. </html>

**Output:**

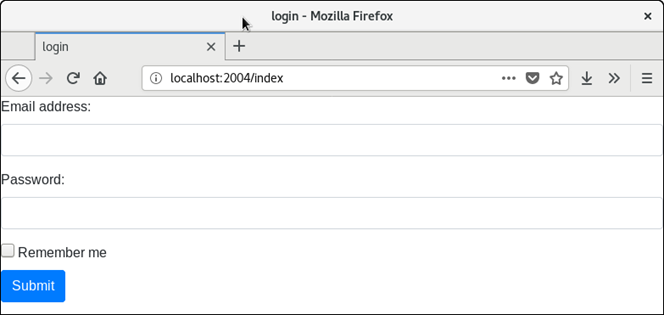


After loading bootstrap files. Our code look like this:

**// index.html**

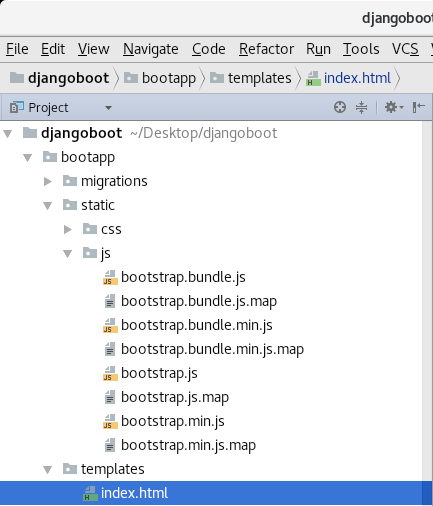
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>login</title>
6. {% load staticfiles %}
7. <link href="{% static 'css/bootstrap.min.css' %}" rel="stylesheet">
8. <script src="{% static 'js/bootstrap.min.js' %}"></script>
9. </head>
10. <body>
11. <form action="/save" method="post">
12. <div **class**="form-group">
13. <label **for**="email">Email address:</label>
14. <input type="email" **class**="form-control" id="email">
15. </div>
16. <div **class**="form-group">
17. <label **for**="pwd">Password:</label>
18. <input type="password" **class**="form-control" id="pwd">
19. </div>
20. <div **class**="checkbox">
21. <label><input type="checkbox"> Remember me</label>
22. </div>
23. <button type="submit" **class**="btn btn-primary">Submit</button>
24. </form>
25. </body>
26. </html>

**Output:**



now, our login form loos much nicer. This is advantage of bootstrap.

Finally, out project structure looks like this.



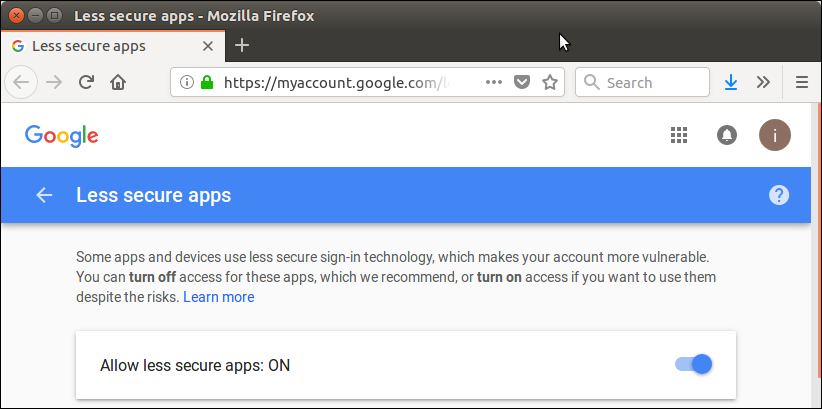
Django Mail Setup

Sending email using Django is pretty easy and require less configuration. In this tutorial, we will send email to provided email.

For this purpose, we will use Google's SMTP and a Gmail account to set sender.

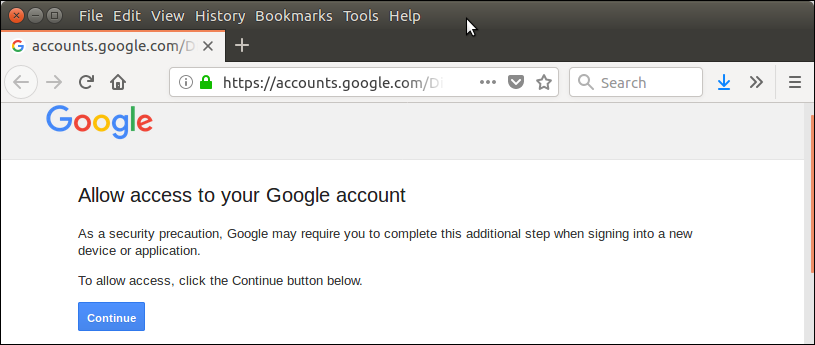
Django provides built-in mail library **django.core.mail** to send email.

Before sending email, we need to make some changes in Gmail account because for security reasons Google does not allow direct access (login) by any application. So, login to the Gmail account and follow the urls. It will redirect to the Gmail account settings where we need to allow less secure apps but toggle the button. See the below screenshot.

<https://myaccount.google.com/lesssecureapps>  


After that follow this url that is a additional security check to verify the make security constraint.

<https://accounts.google.com/DisplayUnlockCaptcha>



Click on continue and all is setup.

Django Configuration

Provide the smtp and Gmail account details into the settings.py file. For example

1. EMAIL\_USE\_TLS = True
2. EMAIL\_HOST = 'smtp.gmail.com'
3. EMAIL\_PORT = 587
4. EMAIL\_HOST\_USER = 'irfan.iit003@gmail.com'
5. EMAIL\_HOST\_PASSWORD = '\*\*\*\*\*\*\*\*\*'

Import Mail Library

1. from django.core.mail **import** send\_mail

Now, write a view function that uses built-in mail function to send mail. See the example

Django Email Example

This example contains the following files.

**// views.py**

1. from django.http **import** HttpResponse
2. from djangpapp **import** settings
3. from django.core.mail **import** send\_mail

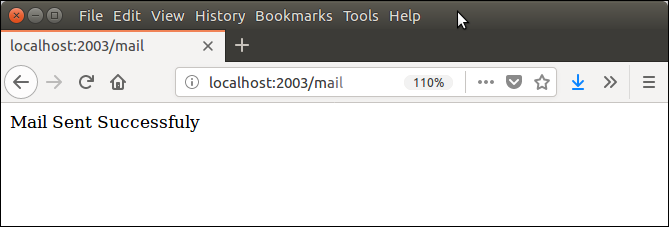
6. def mail(request):
7. subject = "Greetings"
8. msg     = "Congratulations for your success"
9. to      = "irfan.sssit@gmail.com"
10. res     = send\_mail(subject, msg, settings.EMAIL\_HOST\_USER, [to])
11. **if**(res == 1):
12. msg = "Mail Sent Successfuly"
13. **else**:
14. msg = "Mail could not sent"
15. **return** HttpResponse(msg)

**// urls.py**

Put following url into urls.py file.

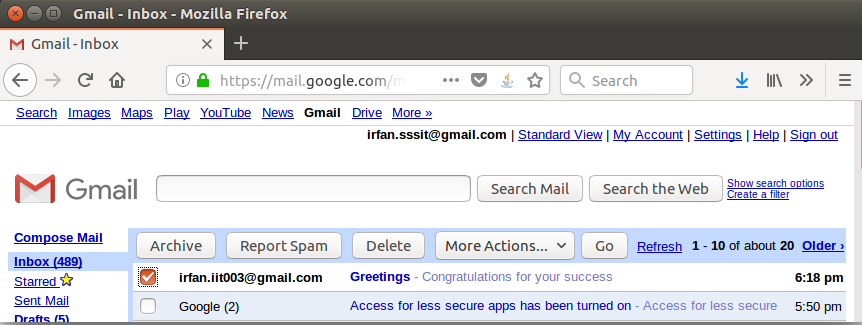
1. path('mail',views.mail)

Run Server and access it at browser, see the output.



Here, the both email ids are mine, so I can verify the email by login to the account.

And after login, here we go!! I got the mail.



Well, same like, we can send mail using other smtp server configurations if we have.

Django Admin

Django provides an admin site to allow CRUD (Create Read Update Delete) operations on registered app model.

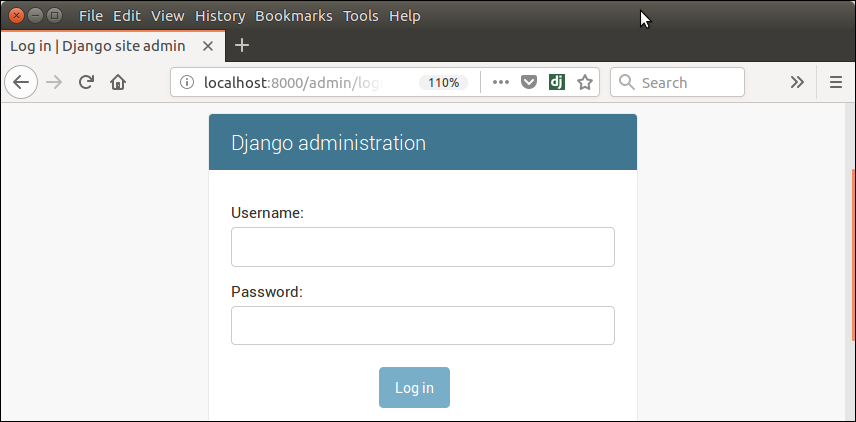
It is a built-in feature of Django that automatically generates interface for models.

We can see the url entry for admin in urls.py file, it is implicit and generated while creating a new project.

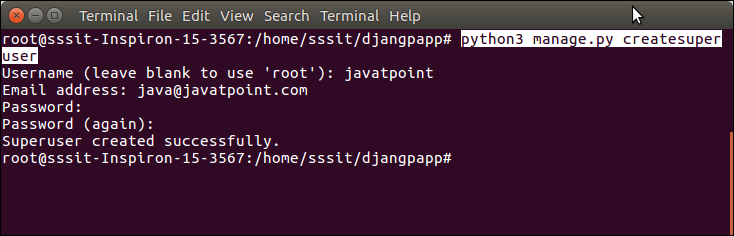
1. urlpatterns = [
2. path('admin/', admin.site.urls),
3. ]

It can be easily accessed by after login from the admin panel, lets run the server **python3 manage.py runserver** and access it through the **localhost:8000/admin.**

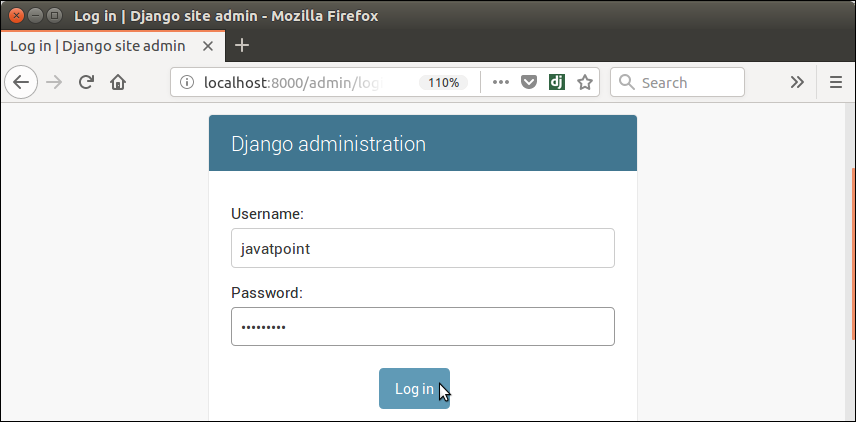
A login form will be displayed, see the below.



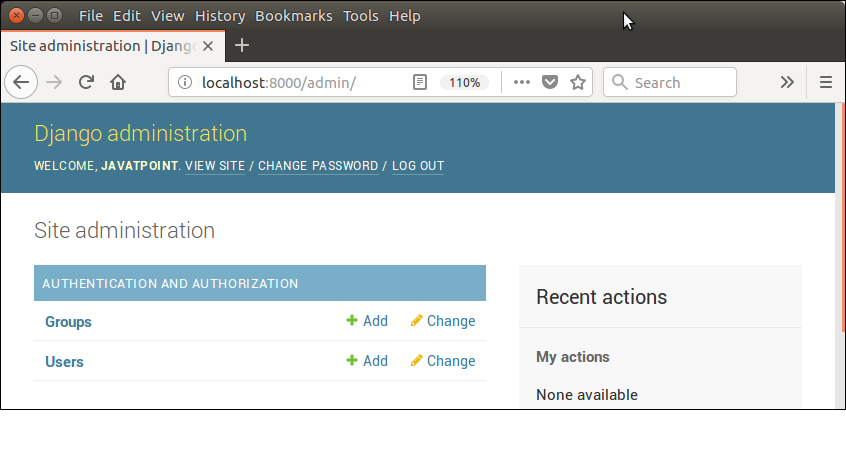
To login, first create admin (super user) user and provide password as we did here:



Super user is created successfully, now login.



It shows a home page after successfully login, see below.



It is an admin dashboard that provides facilities like: creating groups and users. It also used to manage the models.

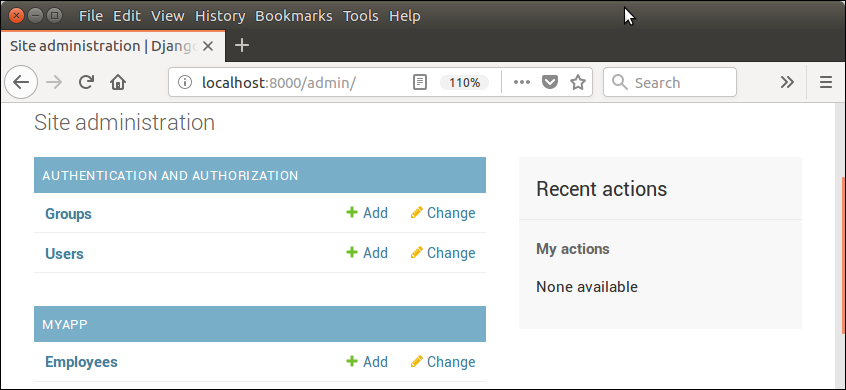
Register Django Model

To register model in **admin.py** file. Use the **admin.site.register()** method and pass the Model name. See the example.

**// admin.py**

1. from django.contrib **import** admin
2. from myapp.models **import** Employee
3. admin.site.register(Employee) # Employee is registered

Login again and see, it has **employee** object.



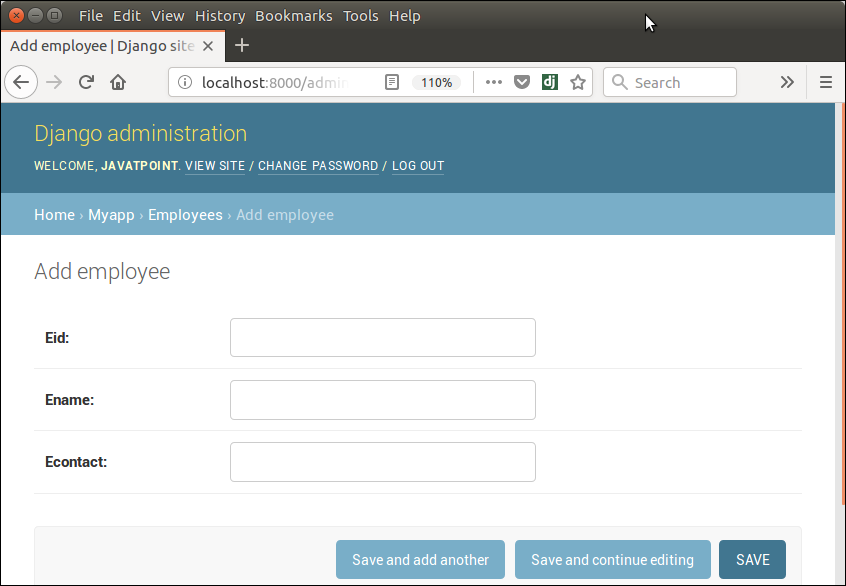
It provides auto generated interface to create new model object. Like, if i click on **add,** it renders a form with all the attributes provided in the model class.

For example, our model class contains the following code.

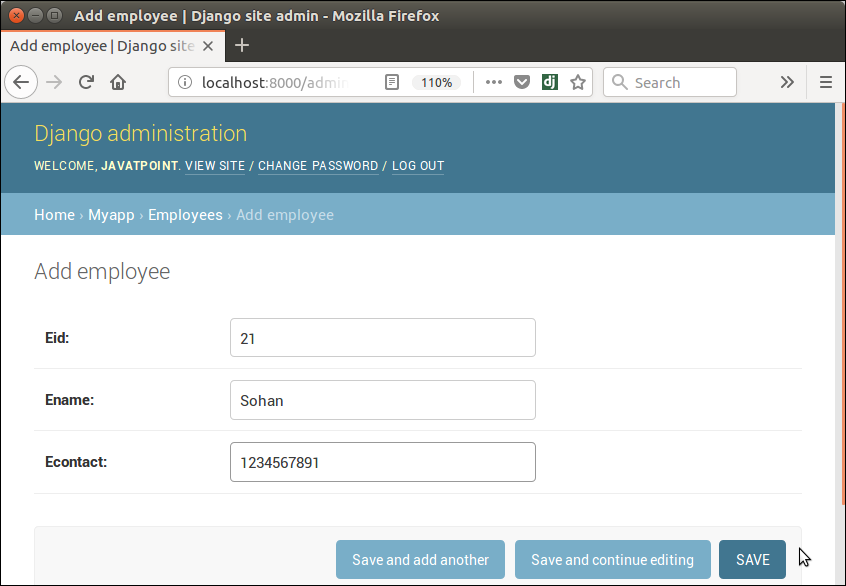
**// models.py**

1. from django.db **import** models
2. **class** Employee(models.Model):
3. eid     = models.CharField(max\_length=20)
4. ename   = models.CharField(max\_length=100)
5. econtact = models.CharField(max\_length=15)
6. **class** Meta:
7. db\_table = "employee"

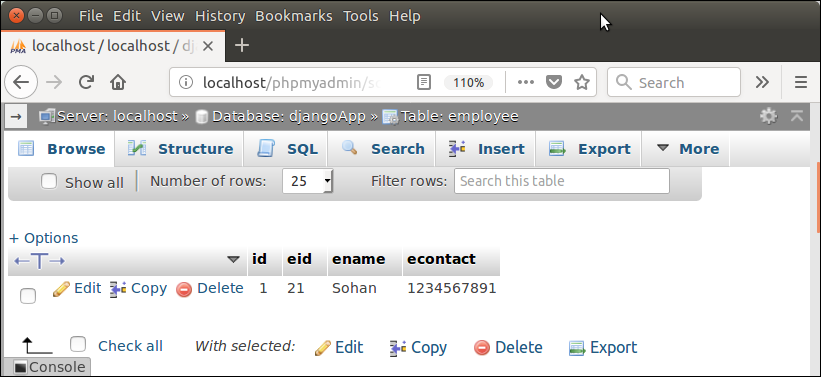
The auto generated form will be based on the model. We don't need to write HTML to create form. The form looks like this:



Lets add an employee by providing details and click on save button.



After saving, record is stored into the database table, see the below MySQL table.



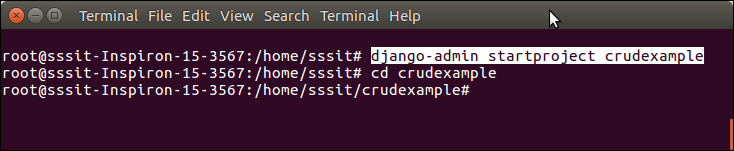
Using this admin dashboard, we can update and delete record also.

Django CRUD (Create Read Update Delete) Example

To create a Django application that performs CRUD operations, follow the following steps.

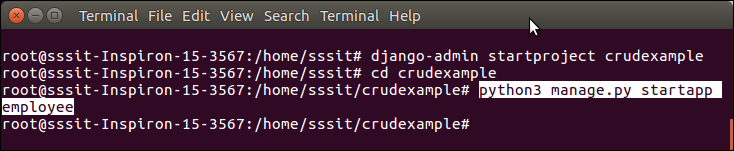
**1. Create a Project**

1. $ django-admin startproject crudexample



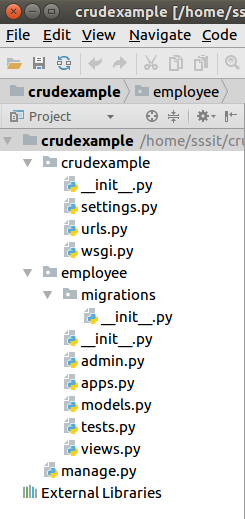
**2. Create an App**

1. $ python3 manage.py startapp employee



**3. Project Structure**

Initially, our project looks like this:



**4. Database Setup**

Create a database **djangodb** in mysql, and configure into the **settings.py** file of django project. See the example.

**// settings.py**

1. DATABASES = {
2. 'default': {
3. 'ENGINE': 'django.db.backends.mysql',
4. 'NAME': 'djangodb',
5. 'USER':'root',
6. 'PASSWORD':'mysql',
7. 'HOST':'localhost',
8. 'PORT':'3306'
9. }
10. }

**5. Create a Model**

Put the following code into **models.py** file.

**// models.py**

1. from django.db **import** models
2. **class** Employee(models.Model):
3. eid = models.CharField(max\_length=20)
4. ename = models.CharField(max\_length=100)
5. eemail = models.EmailField()
6. econtact = models.CharField(max\_length=15)
7. **class** Meta:
8. db\_table = "employee"

**6. Create a ModelForm**

**// forms.py**

1. from django **import** forms
2. from employee.models **import** Employee
3. **class** EmployeeForm(forms.ModelForm):
4. **class** Meta:
5. model = Employee
6. fields = "\_\_all\_\_"

**7. Create View Functions**

**// views.py**

1. from django.shortcuts **import** render, redirect
2. from employee.forms **import** EmployeeForm
3. from employee.models **import** Employee
4. # Create your views here.
5. def emp(request):
6. **if** request.method == "POST":
7. form = EmployeeForm(request.POST)
8. **if** form.is\_valid():
9. **try**:
10. form.save()
11. **return** redirect('/show')
12. except:
13. pass
14. **else**:
15. form = EmployeeForm()
16. **return** render(request,'index.html',{'form':form})
17. def show(request):
18. employees = Employee.objects.all()
19. **return** render(request,"show.html",{'employees':employees})
20. def edit(request, id):
21. employee = Employee.objects.get(id=id)
22. **return** render(request,'edit.html', {'employee':employee})
23. def update(request, id):
24. employee = Employee.objects.get(id=id)
25. form = EmployeeForm(request.POST, instance = employee)
26. **if** form.is\_valid():
27. form.save()
28. **return** redirect("/show")
29. **return** render(request, 'edit.html', {'employee': employee})
30. def destroy(request, id):
31. employee = Employee.objects.get(id=id)
32. employee.delete()
33. **return** redirect("/show")

**8. Provide Routing**

Provide URL patterns to map with views function.

**// urls.py**

1. from django.contrib **import** admin
2. from django.urls **import** path
3. from employee **import** views
4. urlpatterns = [
5. path('admin/', admin.site.urls),
6. path('emp', views.emp),
7. path('show',views.show),
8. path('edit/<int:id>', views.edit),
9. path('update/<int:id>', views.update),
10. path('delete/<int:id>', views.destroy),
11. ]

**9. Organize Templates**

Create a **templates** folder inside the **employee** app and create three (index, edit, show) html files inside the directory. The code for each is given below.

**// index.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. {% load staticfiles %}
7. <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
8. </head>
9. <body>
10. <form method="POST" **class**="post-form" action="/emp">
11. {% csrf\_token %}
12. <div **class**="container">
13. <br>
14. <div **class**="form-group row">
15. <label **class**="col-sm-1 col-form-label"></label>
16. <div **class**="col-sm-4">
17. <h3>Enter Details</h3>
18. </div>
19. </div>
20. <div **class**="form-group row">
21. <label **class**="col-sm-2 col-form-label">Employee Id:</label>
22. <div **class**="col-sm-4">
23. {{ form.eid }}
24. </div>
25. </div>
26. <div **class**="form-group row">
27. <label **class**="col-sm-2 col-form-label">Employee Name:</label>
28. <div **class**="col-sm-4">
29. {{ form.ename }}
30. </div>
31. </div>
32. <div **class**="form-group row">
33. <label **class**="col-sm-2 col-form-label">Employee Email:</label>
34. <div **class**="col-sm-4">
35. {{ form.eemail }}
36. </div>
37. </div>
38. <div **class**="form-group row">
39. <label **class**="col-sm-2 col-form-label">Employee Contact:</label>
40. <div **class**="col-sm-4">
41. {{ form.econtact }}
42. </div>
43. </div>
44. <div **class**="form-group row">
45. <label **class**="col-sm-1 col-form-label"></label>
46. <div **class**="col-sm-4">
47. <button type="submit" **class**="btn btn-primary">Submit</button>
48. </div>
49. </div>
50. </div>
51. </form>
52. </body>
53. </html>

**// show.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Employee Records</title>
6. {% load staticfiles %}
7. <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
8. </head>
9. <body>
10. <table **class**="table table-striped table-bordered table-sm">
11. <thead **class**="thead-dark">
12. <tr>
13. <th>Employee ID</th>
14. <th>Employee Name</th>
15. <th>Employee Email</th>
16. <th>Employee Contact</th>
17. <th>Actions</th>
18. </tr>
19. </thead>
20. <tbody>
21. {% **for** employee in employees %}
22. <tr>
23. <td>{{ employee.eid }}</td>
24. <td>{{ employee.ename }}</td>
25. <td>{{ employee.eemail }}</td>
26. <td>{{ employee.econtact }}</td>
27. <td>
28. <a href="/edit/{{ employee.id }}"><span **class**="glyphicon glyphicon-pencil" >Edit</span></a>
29. <a href="/delete/{{ employee.id }}">Delete</a>
30. </td>
31. </tr>
32. {% endfor %}
33. </tbody>
34. </table>
35. <br>
36. <br>
37. <center><a href="/emp" **class**="btn btn-primary">Add New Record</a></center>
38. </body>
39. </html>

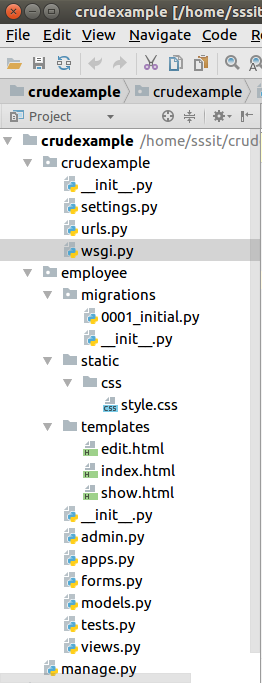
**// edit.html**

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. {% load staticfiles %}
7. <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
8. </head>
9. <body>
10. <form method="POST" **class**="post-form" action="/update/{{employee.id}}">
11. {% csrf\_token %}
12. <div **class**="container">
13. <br>
14. <div **class**="form-group row">
15. <label **class**="col-sm-1 col-form-label"></label>
16. <div **class**="col-sm-4">
17. <h3>Update Details</h3>
18. </div>
19. </div>
20. <div **class**="form-group row">
21. <label **class**="col-sm-2 col-form-label">Employee Id:</label>
22. <div **class**="col-sm-4">
23. <input type="text" name="eid" id="id\_eid" required maxlength="20" value="{{ employee.eid }}"/>
24. </div>
25. </div>
26. <div **class**="form-group row">
27. <label **class**="col-sm-2 col-form-label">Employee Name:</label>
28. <div **class**="col-sm-4">
29. <input type="text" name="ename" id="id\_ename" required maxlength="100" value="{{ employee.ename }}" />
30. </div>
31. </div>
32. <div **class**="form-group row">
33. <label **class**="col-sm-2 col-form-label">Employee Email:</label>
34. <div **class**="col-sm-4">
35. <input type="email" name="eemail" id="id\_eemail" required maxlength="254" value="{{ employee.eemail }}" />
36. </div>
37. </div>
38. <div **class**="form-group row">
39. <label **class**="col-sm-2 col-form-label">Employee Contact:</label>
40. <div **class**="col-sm-4">
41. <input type="text" name="econtact" id="id\_econtact" required maxlength="15" value="{{ employee.econtact }}" />
42. </div>
43. </div>
44. <div **class**="form-group row">
45. <label **class**="col-sm-1 col-form-label"></label>
46. <div **class**="col-sm-4">
47. <button type="submit" **class**="btn btn-success">Update</button>
48. </div>
49. </div>
50. </div>
51. </form>
52. </body>
53. </html>

**10. Static Files Handling**

Create a folder **static/css** inside the **employee** app and put a css inside it. Download the css file here [Click Here.](https://www.javatpoint.com/style.css)

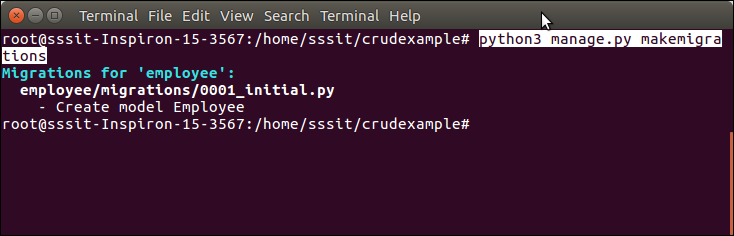
**11. Project Structure**



**12. Create Migrations**

Create migrations for the created model employee, use the following command.

1. $ python3 manage.py makemigrations



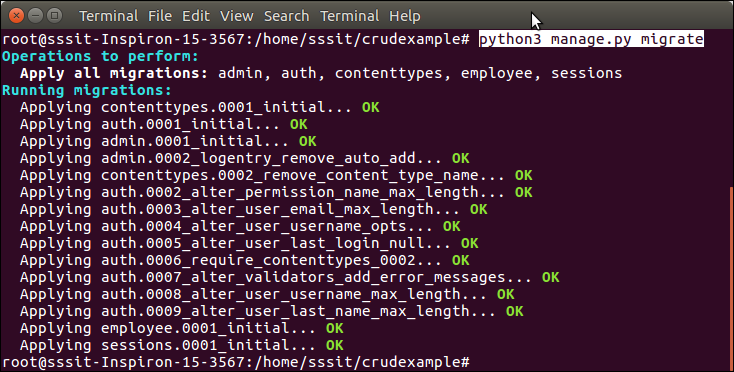
After migrations, execute one more command to reflect the migration into the database. But before it, mention name of app (employee) in INSTALLED\_APPS of settings.py file.

**// settings.py**

1. INSTALLED\_APPS = [
2. 'django.contrib.admin',
3. 'django.contrib.auth',
4. 'django.contrib.contenttypes',
5. 'django.contrib.sessions',
6. 'django.contrib.messages',
7. 'django.contrib.staticfiles',
8. 'employee'
9. ]

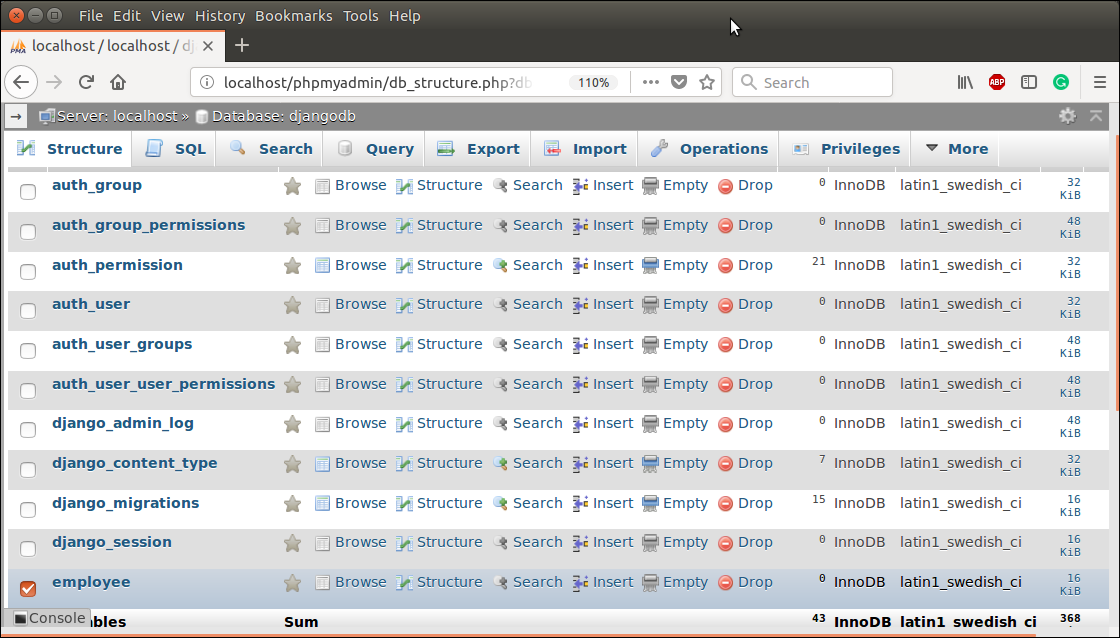
Run the command to migrate the migrations.

1. $ python3 manage.py migrate



Now, our application has successfully connected and created tables in database. It creates 10 default tables for handling project (session, authentication etc) and one table of our model that we created.

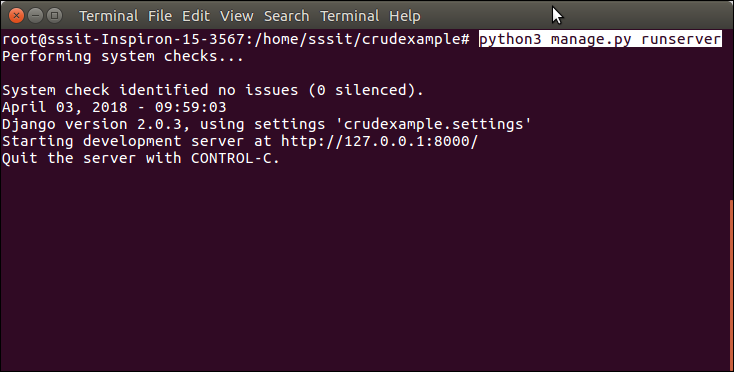
See list of tables created after migrate command.



Run Server

To run server use the following command.

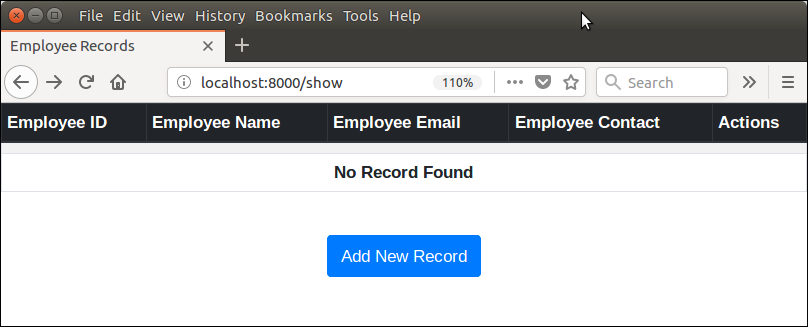
1. $ python3 manage.py runserver



Access to the Browser

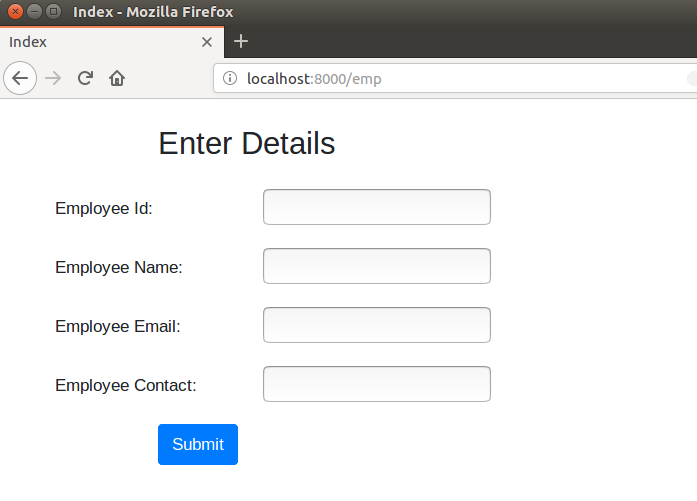
Access the application by entering **localhost:8000/show**, it will show all the available employee records.

Initially, there is no record. So, it shows no record message.



Adding Record

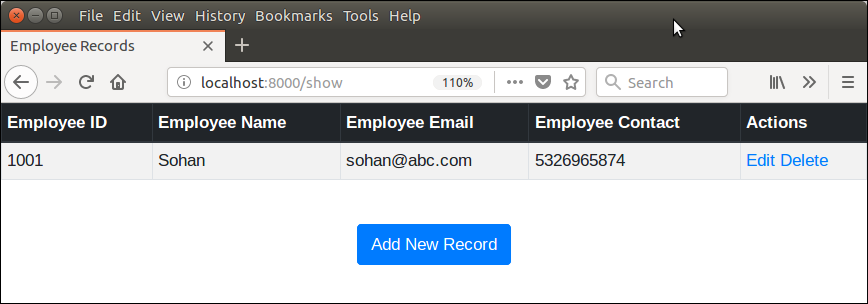
Click on the **Add New Record** button and fill the details. See the example.



Filling the details.



Submit the record and see, after submitting it shows the saved record.

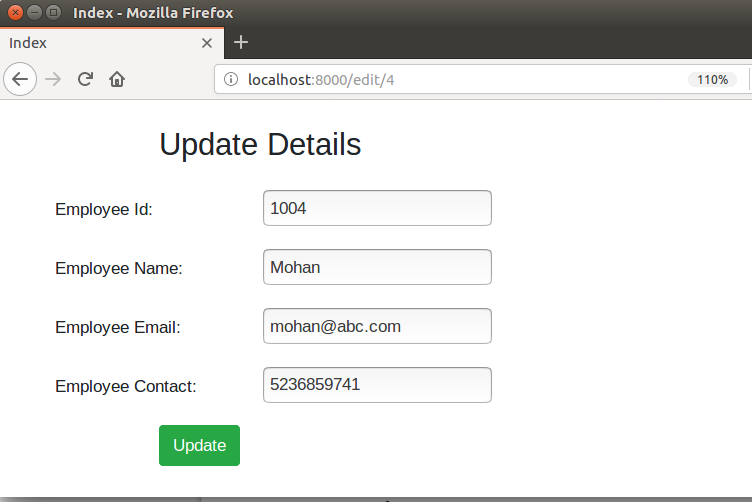


This section also allows, update and delete records from the **actions** column.

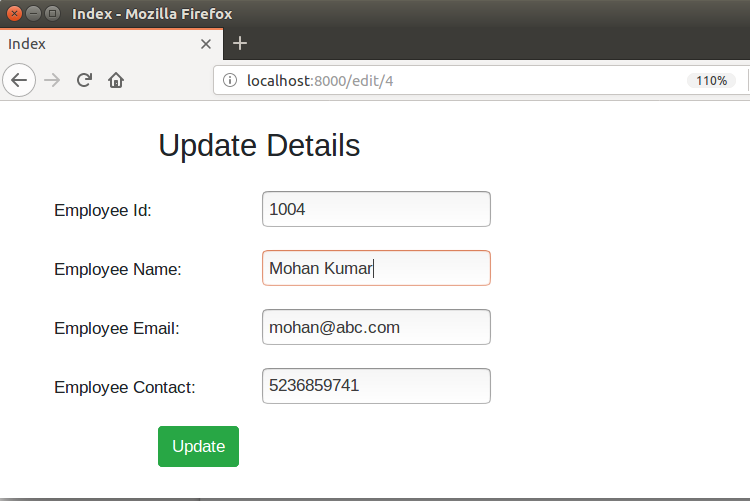
After saving couple of records, now we have following records.

Update Record

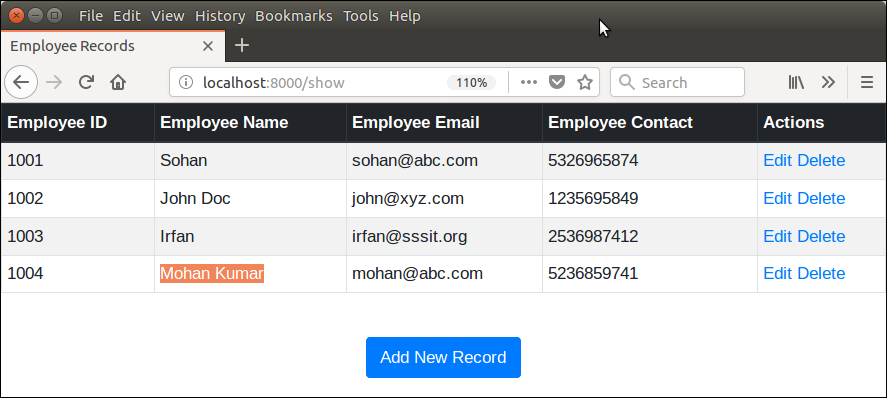
Lets update the record of **Mohan** by clicking on **edit** button. It will display record of Mohan in edit mode.



Lets, suppose I update **mohan** to **mohan kumar** then click on the update button. It updates the record immediately. See the example.



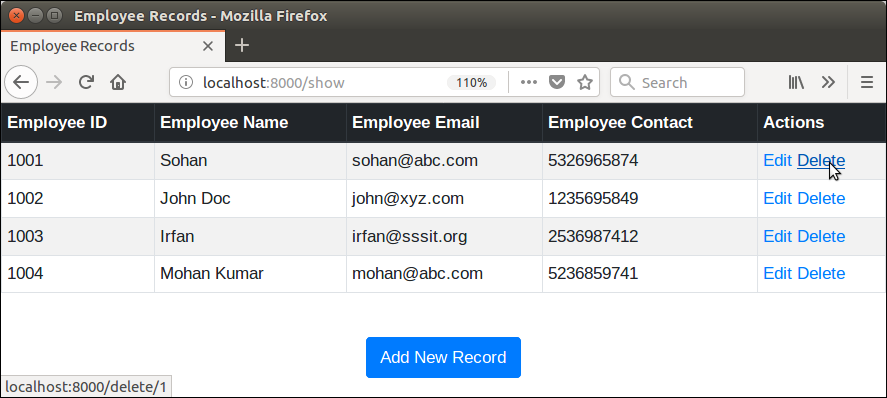
Click on update button and it redirects to the following page. See name is updated.



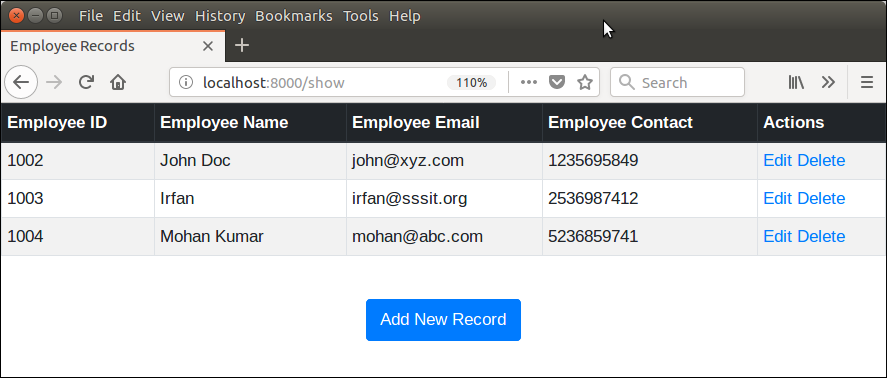
Same like, we can delete records too, by clicking the **delete** link.

Delete Record

Suppose, I want to delete **Sohan**, it can be done easily by clicking the delete button. See the example.



After deleting, we left with the following records.



Well, we have successfully created a CRUD application using Django.