DJANGO NOTES FOR PROFESSIONALS

Made by – Deeproshan Kumar

Introduction

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

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

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

History

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

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

Django Version History

Version	Date	Description
0.90	16 Nov 2005	
0.91	11 Jan 2006	magic removal
0.96	23 Mar 2007	newforms, testing tools
1.0	3 Sep 2008	API stability, decoupled admin, unicode
1.1	29 Jul 2009	Aggregates, transaction based tests
1.2	17 May 2010	Multiple db connections, CSRF, model validation
1.3	23 Mar 2011	Timezones, in browser testing, app templates.
1.5	26 Feb 2013	Python 3 Support, configurable user model
1.6	6 Nov 2013	Dedicated to Malcolm Tredinnick, db transaction management, connection pooling.
1.7	2 Sep 2014	Migrations, application loading and configuration.
1.8 LTS	2 Sep 2014	Migrations, application loading and configuration.
1.8 LTS	1 Apr 2015	Native support for multiple template engines.*Supported until at least April 2018*

1.9	1 Dec 2015	Automatic password validation. New styling for admin interface.
1.10	1 Aug 2016	Full text search for PostgreSQL. New-style middleware.
1.11 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 make a framework which takes less time to build web application. The project implementation phase is a very time taken but Django creates it rapidly.

Secure

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

Scalable

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

Fully loaded

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

Versatile

Django is versatile in nature which allows it to build applications for different-different domains. Now a days, Companies are using Django to build various types of applications like: content management systems, social networks sites or scientific computing platforms etc.

Open Source

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

Vast and Supported Community

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

Django Installation

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

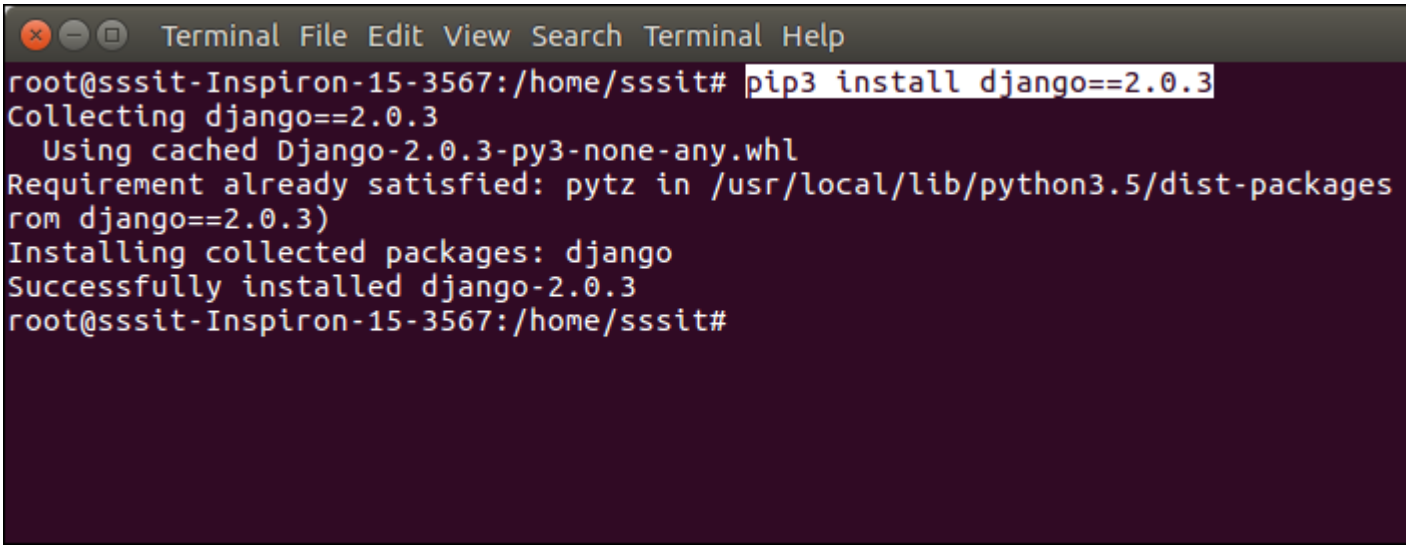
Django requires **pip** to start installation. Pip is a package manager system which is used to install and manage packages written in python. For Python 3.4 and higher versions **pip3** is used to manage packages.

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

The complete installation process is described below. Before installing make sure **pip is installed** in local system.

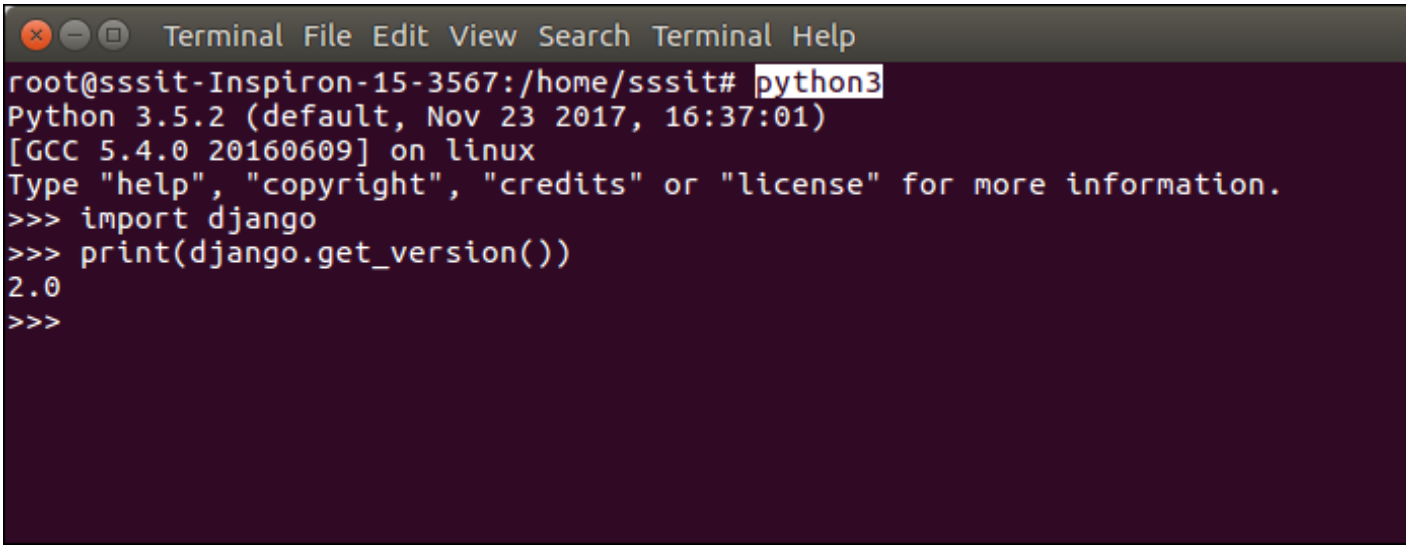
Here, we are installing Django using pip3, the installation command is given below.

$ pip3 install django==2.0.3



Verify Django Installation

After installing Django, we need to varify the installation. Open terminal and write **python3** and press enter. It will display python shell where we can verify django installation.



Look at the Django version displayed by the print method of the python. Well, Django is installed successfully. Now, we can build Django web applications.

Django Project

In the previous topic, we have installed Django successfully. Now, we will learn step by step process to create a Django application.

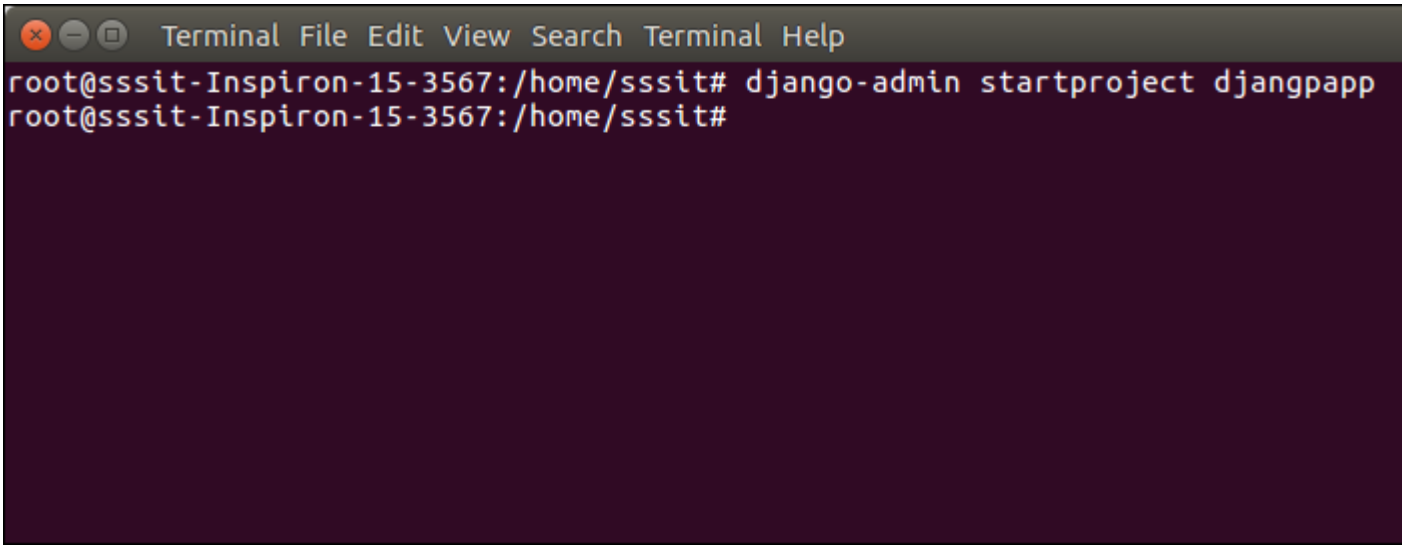
To create a Django project, we can use the following command. *projectname* is the name of Django application.

$ django-admin startproject projectname

Django Project Example

Here, we are creating a project **djangpapp** in the current directory.

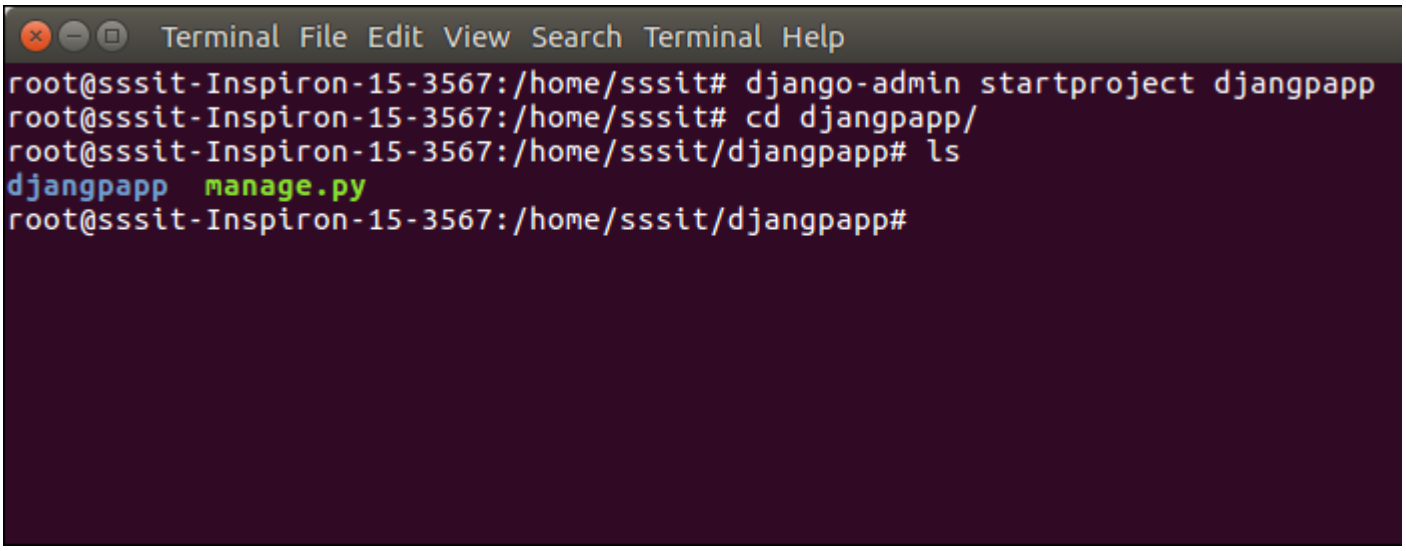
$ django-admin startproject djangpapp



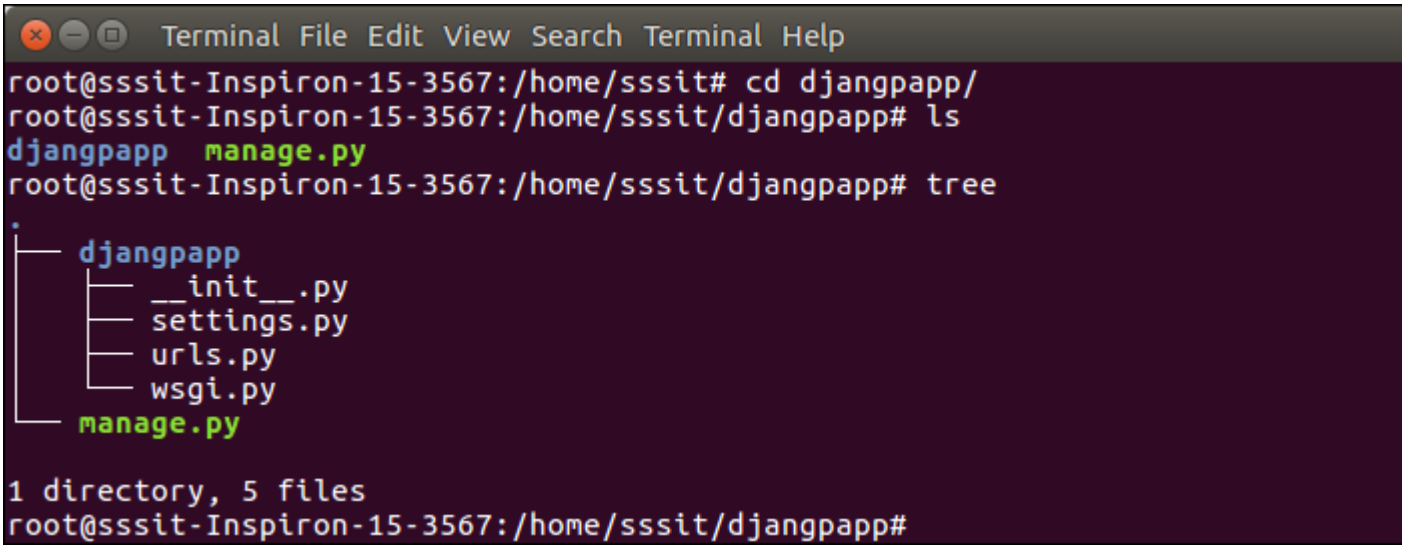
Locate into the Project

Now, move to the project by changing the directory. The Directory can be changed by using the following command.

cd djangpapp



To see all the files and subfolders of django project, we can use **tree** command to view the tree structure of the application. This is a utility command, if it is not present, can be downloaded via **apt-get install tree** command.



A Django project 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 the project in various ways and also used to manage an 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 module inside the application.
- **__init__.py:** It is an empty file that tells to the Python that this directory should be considered as a Python package.
- **settings.py:** This file is used to configure application settings such as database connection, static files linking etc.
- **urls.py:** This file contains the listed URLs of the application. In this file, we can mention the URLs and corresponding actions to perform the task and display the view.
- **wsgi.py:** It is an entry-point for WSGI-compatible web servers to serve Django project.

Initially, this project is a default draft which contains all the required files and folders.

Running the Django Project

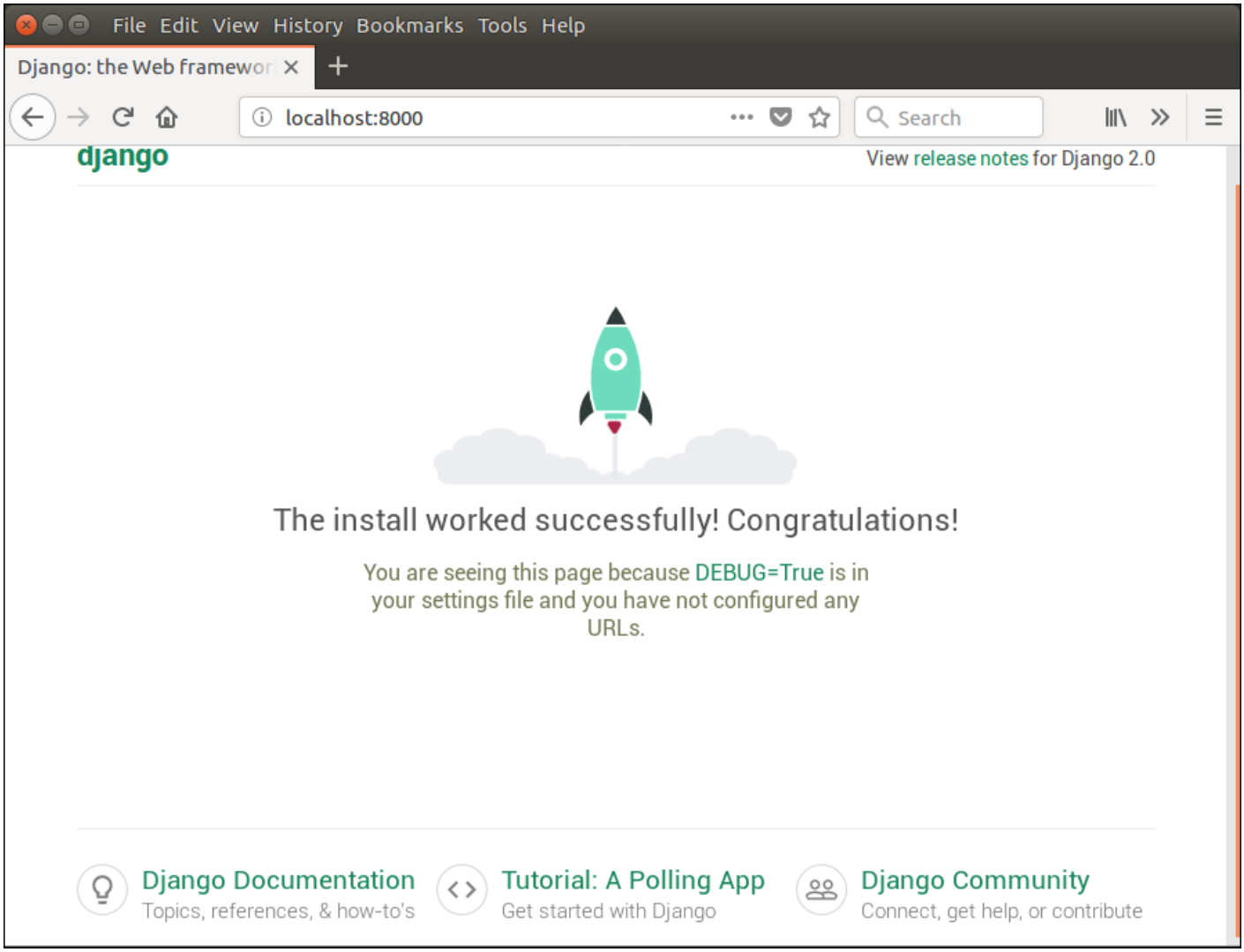
Django project has a built-in development server which is used to run application instantly without any external web server. It means we don't need of Apache or another web server to run the application in development mode.

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

$ python3 manage.py runserver

```
Terminal File Edit View Search Terminal Help
root@sssit-Inspiron-15-3567:/home/sssit/djangpapp# python3 manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have 14 unapplied migration(s). Your project may not work properly until you
 apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

March 13, 2018 - 07:21:03
Django version 2.0, using settings 'djangpapp.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

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



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

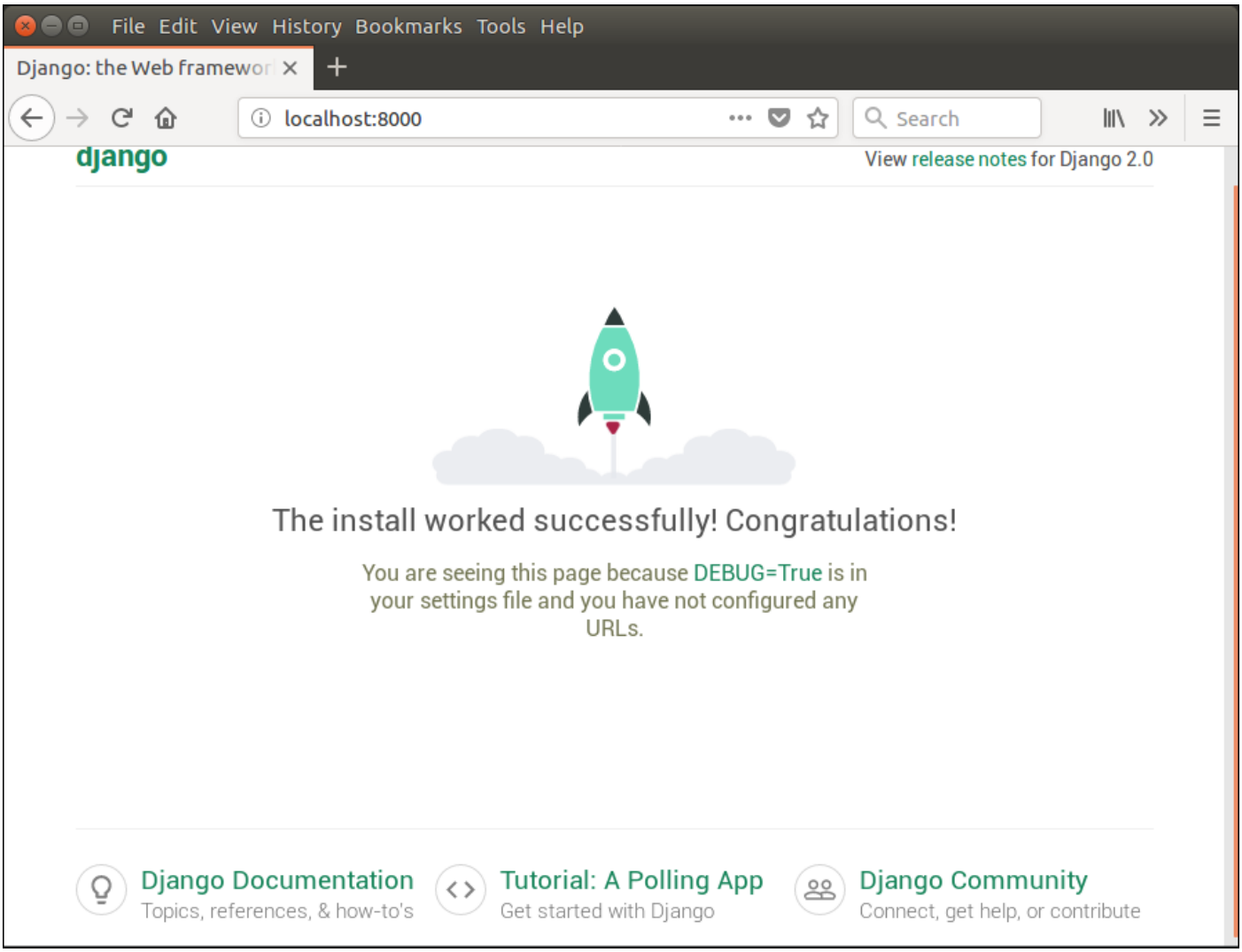
Django Configuration with Apache Web Server

Django uses its built-in development server to run the web application. To start this server, we can use **python manage.py runserver** command.

This command starts the server which runs on port 8000 and can be accessed at browser by entering *localhost:8000*. It shows a welcome page of the application.

```
Terminal  File  Edit  View  Search  Terminal  Help
root@sssit-Inspiron-15-3567:/home/sssit/djangpapp# python3 manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have 14 unapplied migration(s). Your project may not work properly until you
 apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

March 13, 2018 - 07:21:03
Django version 2.0, using settings 'djangpapp.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

And at browser, it can be accessed as below.

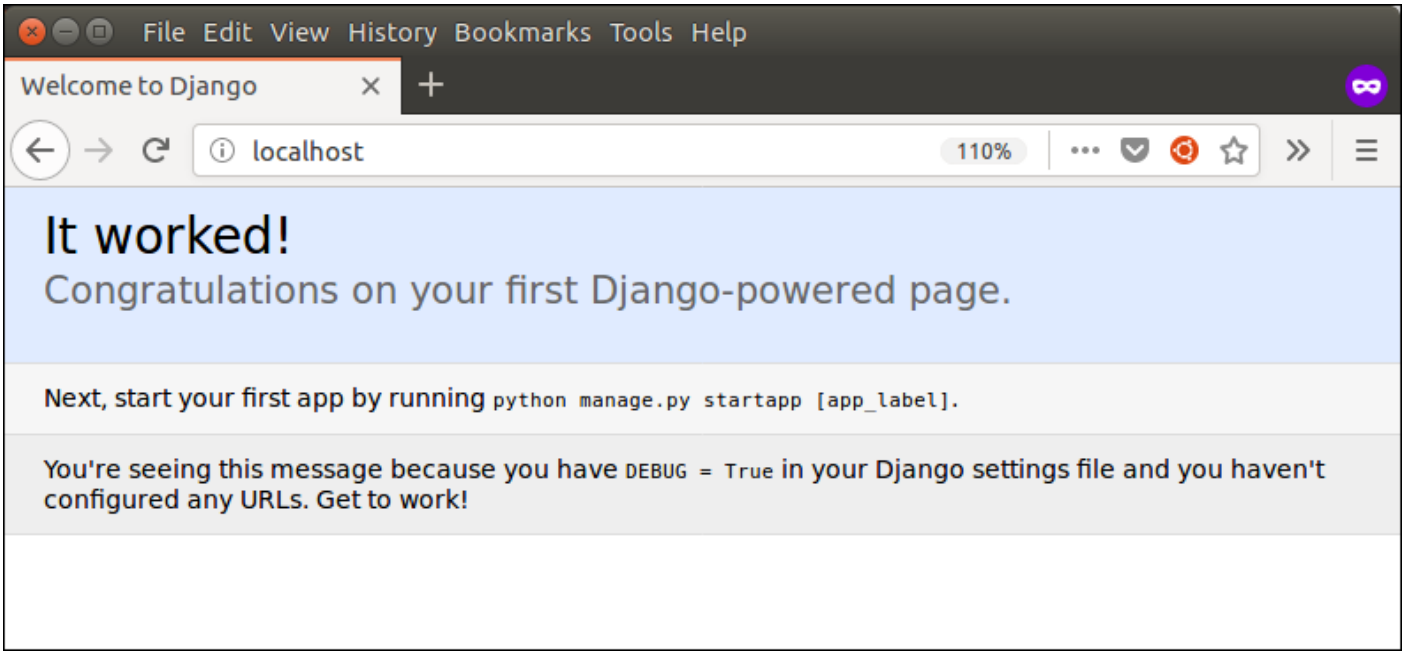


But if we want to run our application by using **apache server** rather than built-in development server, we need to configure **apache2.conf** file located at /**etc**/**apache** directory. Add the following code into this file.

// apache2.conf

```
WSGIScriptAlias / /var/www/html/django7/django7/wsgi.py
WSGIPythonPath /var/www/html/django7/

<Directory /var/www/html/django7>
  <Files wsgi.py>
          Require all granted
  </Files>
</Directory>
```

After adding these lines, restart apache server by using the **service apache2 restart** command and then type **localhost** to the browser's address bar. This time, project will run on apache server rather than a built-in server. See, it shows the home page of the application.



Django Virtual Environment Setup

The virtual environment is an environment which is used by Django to execute an application. It is recommended to create and execute a Django application in a separate environment. Python provides a tool **virtualenv** to create an isolated Python environment. We will use this tool to create a virtual environment for our Django application.

To set up a virtual environment, use the following steps.

1. Install Package

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

$ apt-get install python3-venv

```
Terminal File Edit View Search Terminal Help
root@sssit-Inspiron-15-3567:/home/sssit# apt-get install python3-venv
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  python3.5-venv
The following NEW packages will be installed:
  python3-venv python3.5-venv
0 upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
Need to get 7,104 B of archives.
After this operation, 39.9 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://archive.ubuntu.com/ubuntu xenial-updates/universe amd64 python3.5-v
env amd64 3.5.2-2ubuntu0~16.04.4 [5,998 B]
Get:2 http://archive.ubuntu.com/ubuntu xenial/universe amd64 python3-venv amd64
3.5.1-3 [1,106 B]
Fetched 7,104 B in 1s (4,909 B/s)
Selecting previously unselected package python3.5-venv.
```

2. Create a Directory

$ mkdir djangoenv

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

```
root@sssit-Inspiron-15-3567: /home/sssit/djangoenv
root@sssit-Inspiron-15-3567:/home/sssit# mkdir djangoenv
root@sssit-Inspiron-15-3567:/home/sssit# cd djangoenv/
root@sssit-Inspiron-15-3567:/home/sssit/djangoenv#
```

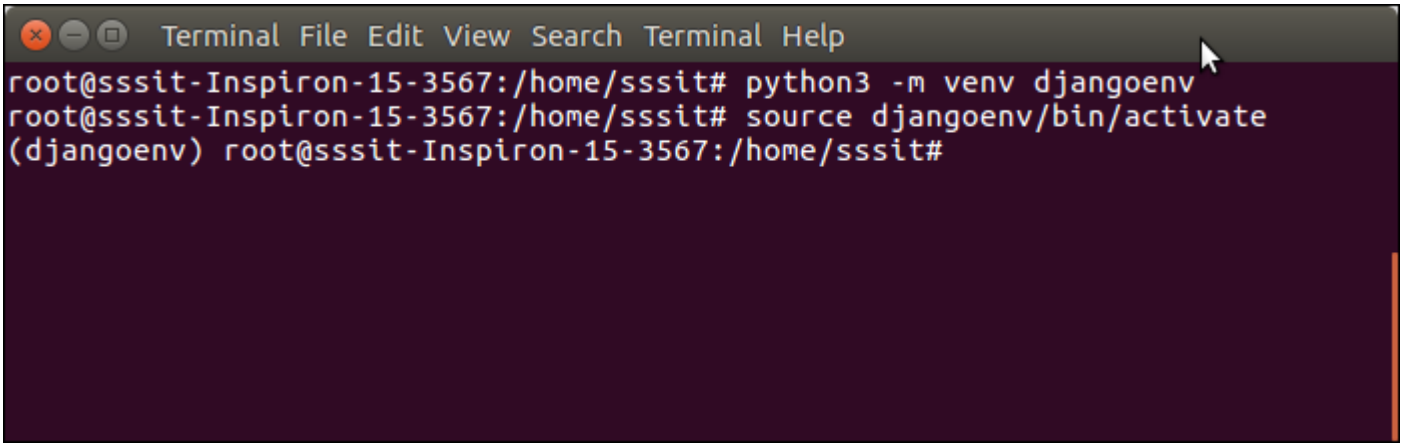
3. Create Virtual Environment

$ python3 -m venv djangoenv

4. Activate Virtual Environment

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

$ source djangoenv/bin/activate

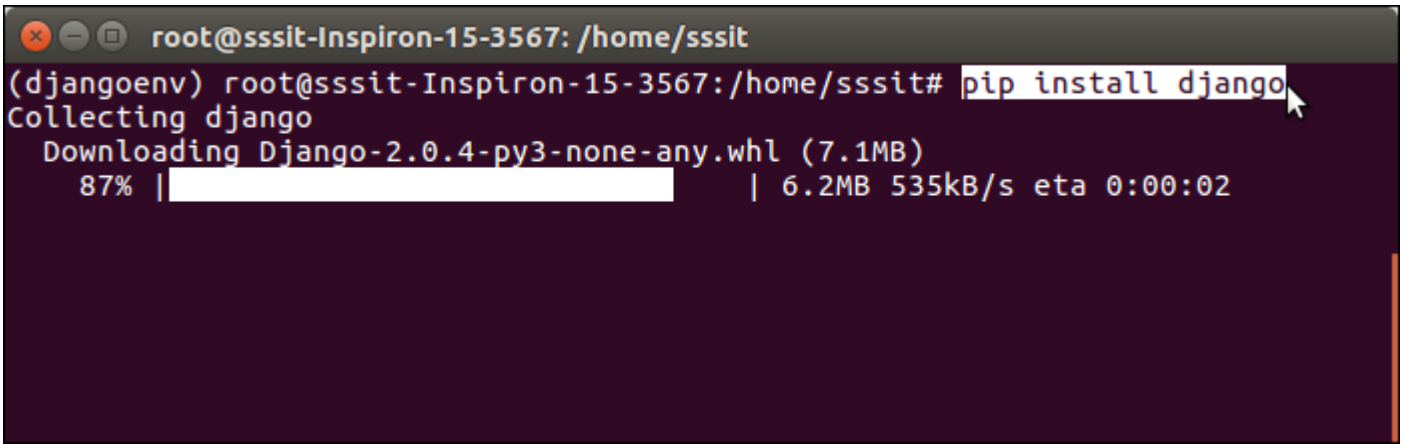


Till here, the virtual environment has started. Now, we can use it to create Django application.

Install Django

Install Django in the virtual environment. To install Django, use the following command.

$ pip install django



Django has installed successfully. Now we can create a new project and build new applications in the separate environment.

Django Admin Interface

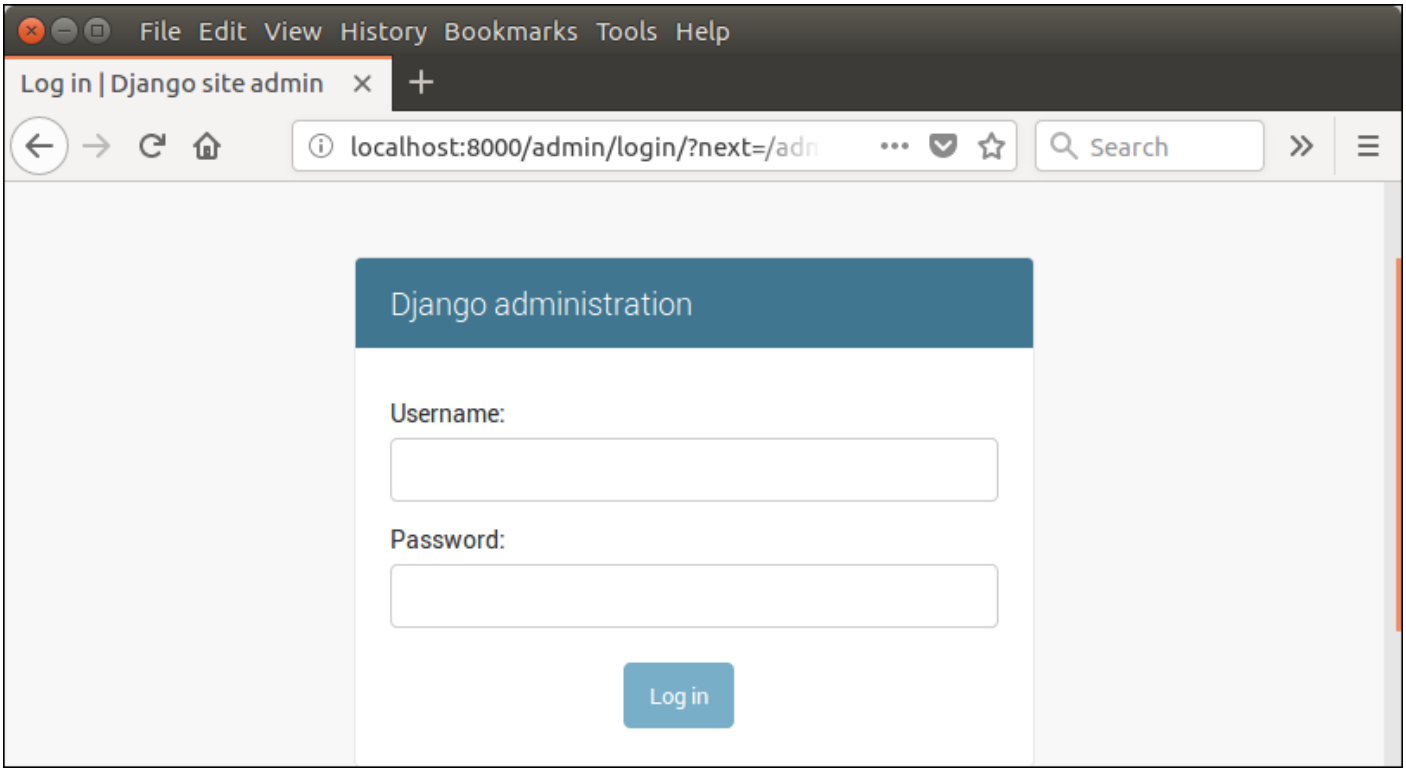
Django provides a built-in admin module which can be used to perform CRUD operations on the models. It reads metadata from the model to provide a quick interface where the user can manage the content of the application.

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

Let's see how to activate and use Django's admin module (interface).

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

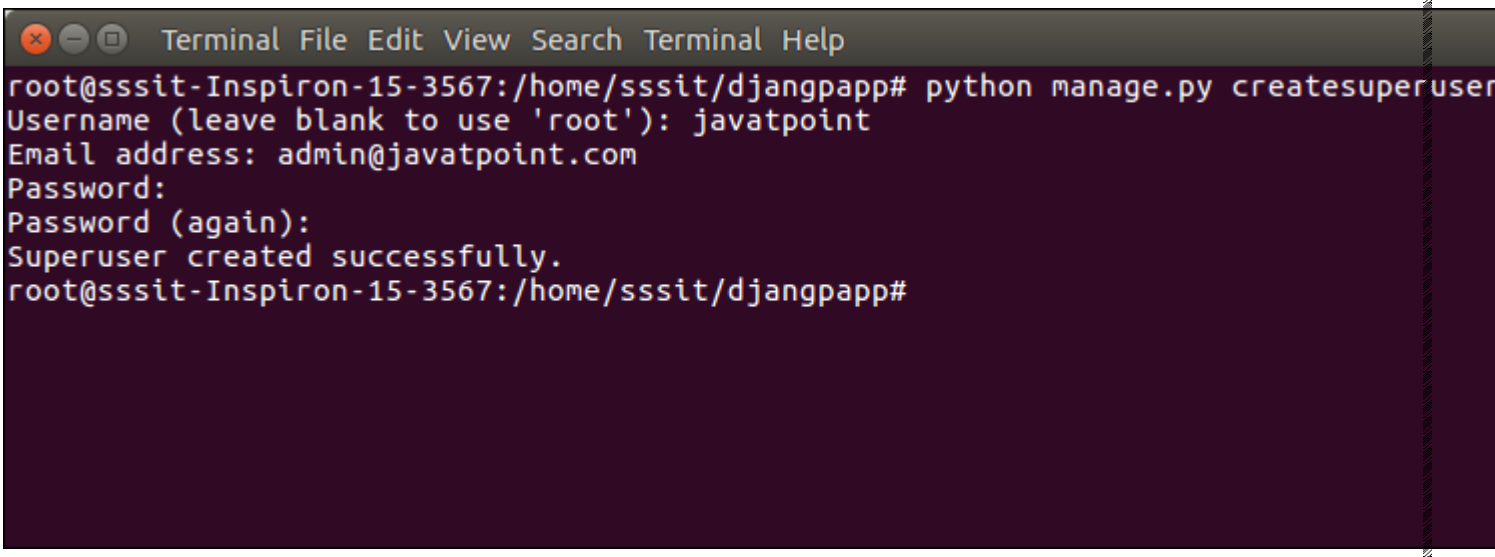
To access it at browser use '/**admin**/' at a local machine like **localhost:8000**/**admin**/ and it shows the following output:



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

Create an Admin User

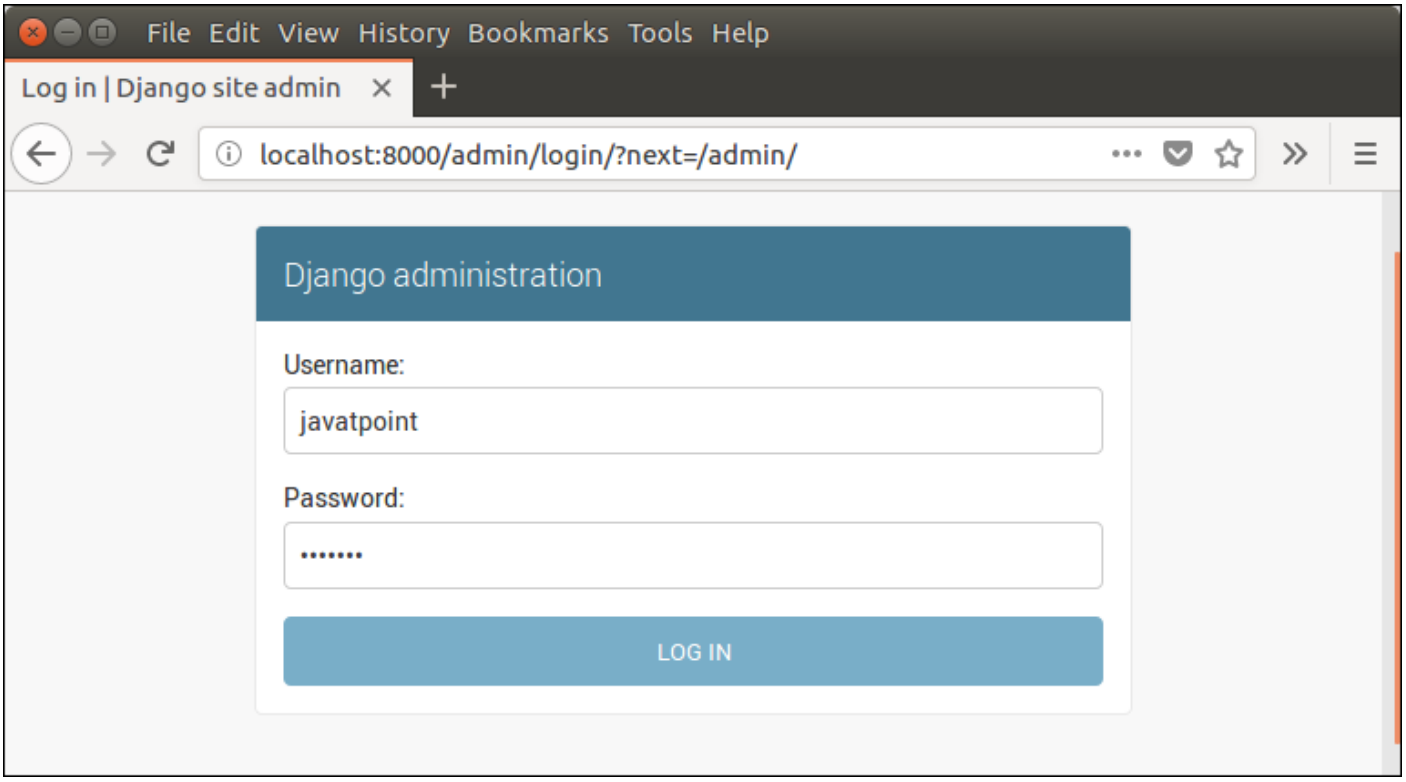
$ python3 managen.py createsuperuser



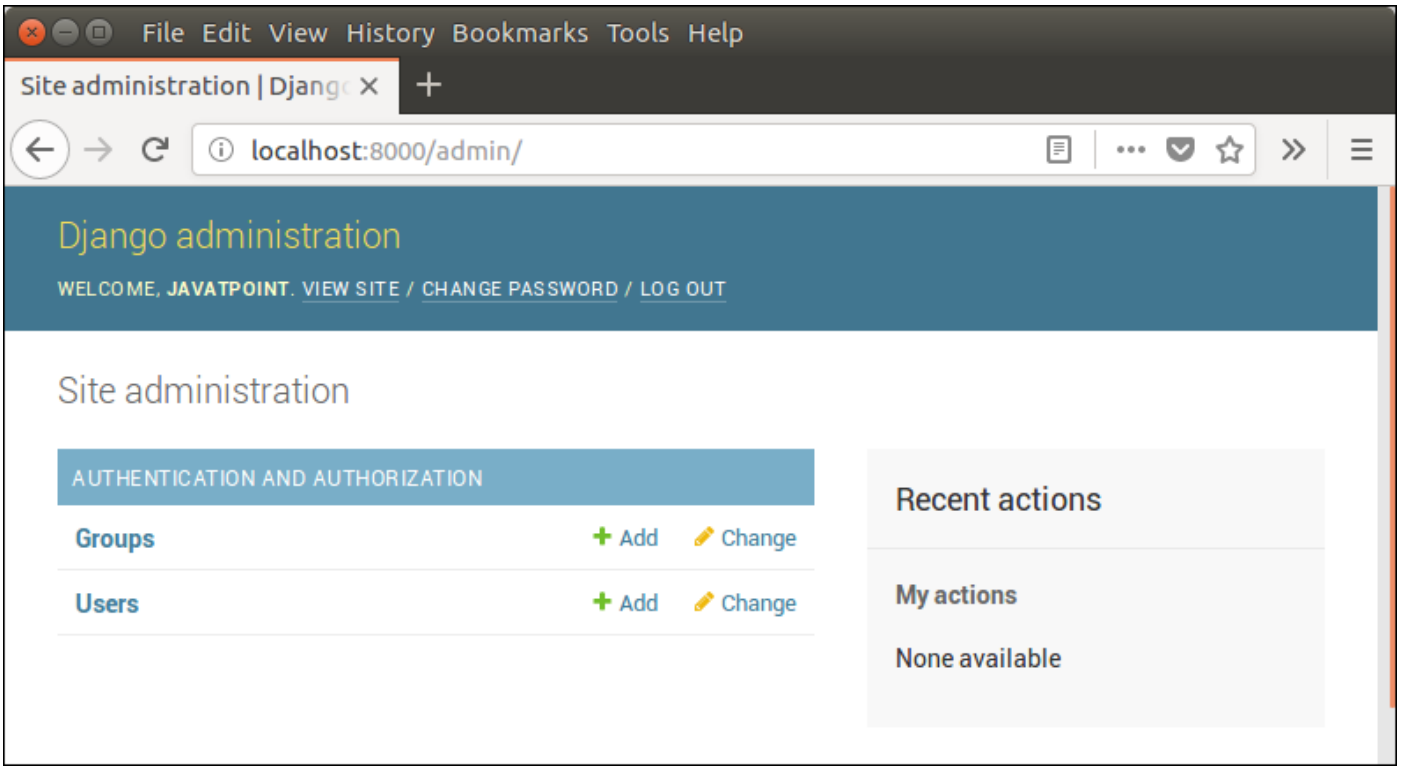
Now start development server and access admin login.

$ 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 models. The model registration process will be discussed in further chapters.

Django App

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

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

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

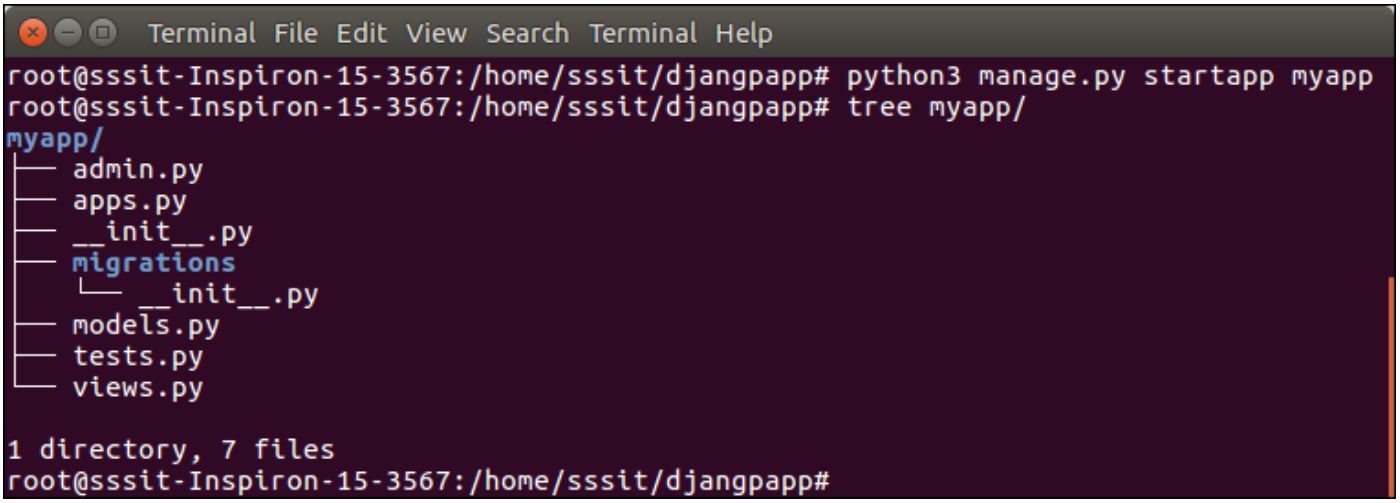
Creating an App

To create an app, we can use the following command.

$ python3 manage.py startapp appname

Django App Example

$ python3 manage.py startapp myapp



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

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

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

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

// views.py

```
from django.shortcuts import render

# Create your views here.
from django.http import HttpResponse

def hello(request):
    return HttpResponse("<h2>Hello, Welcome to Django!</h2>")
```

// urls.py

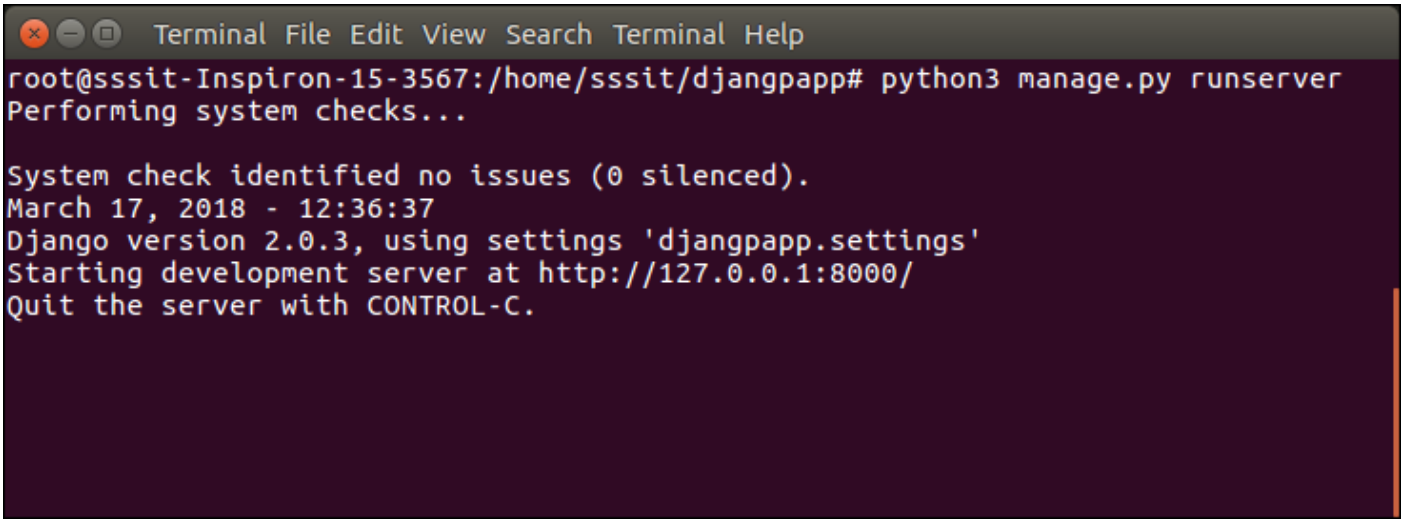
```
from django.contrib import admin
from django.urls import path
```

```
from myapp import views

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

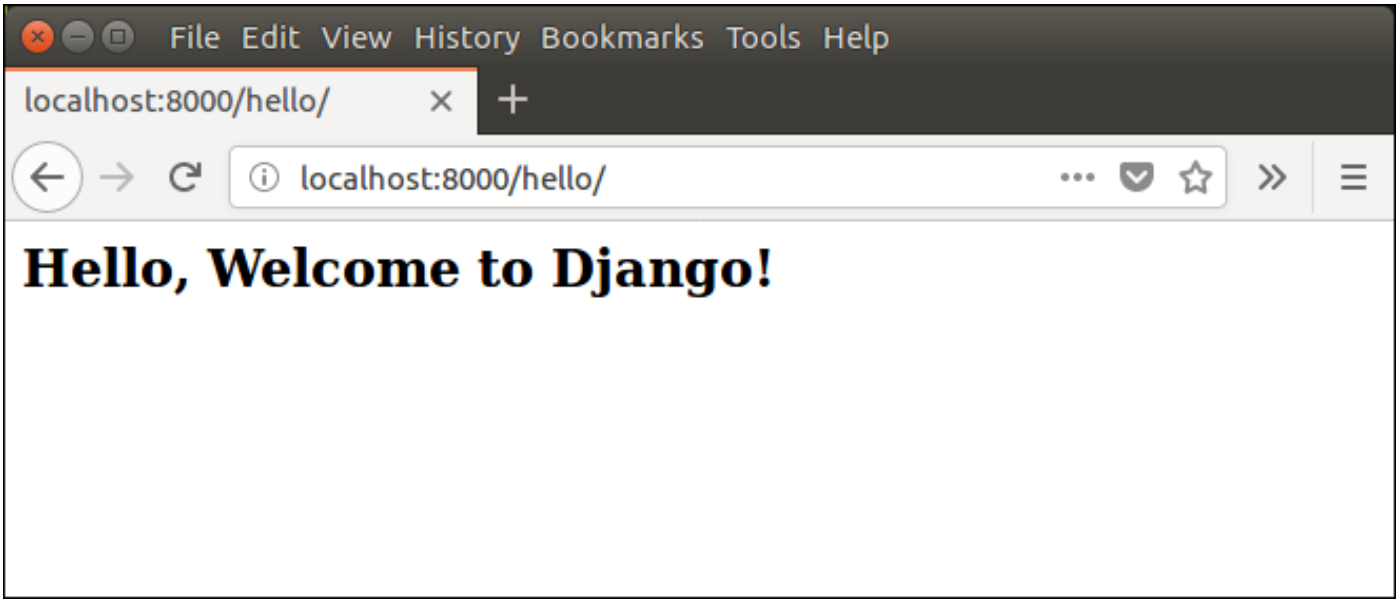
We have made changes in two files of the application. Now, let's run the it by using the following command. This command will start the server at port 8000.

Run the Application

$ python3 manage.py runserver



Open any web browser and enter the URL **localhost:8000/hello**. It will show the output given below.



Django MVT

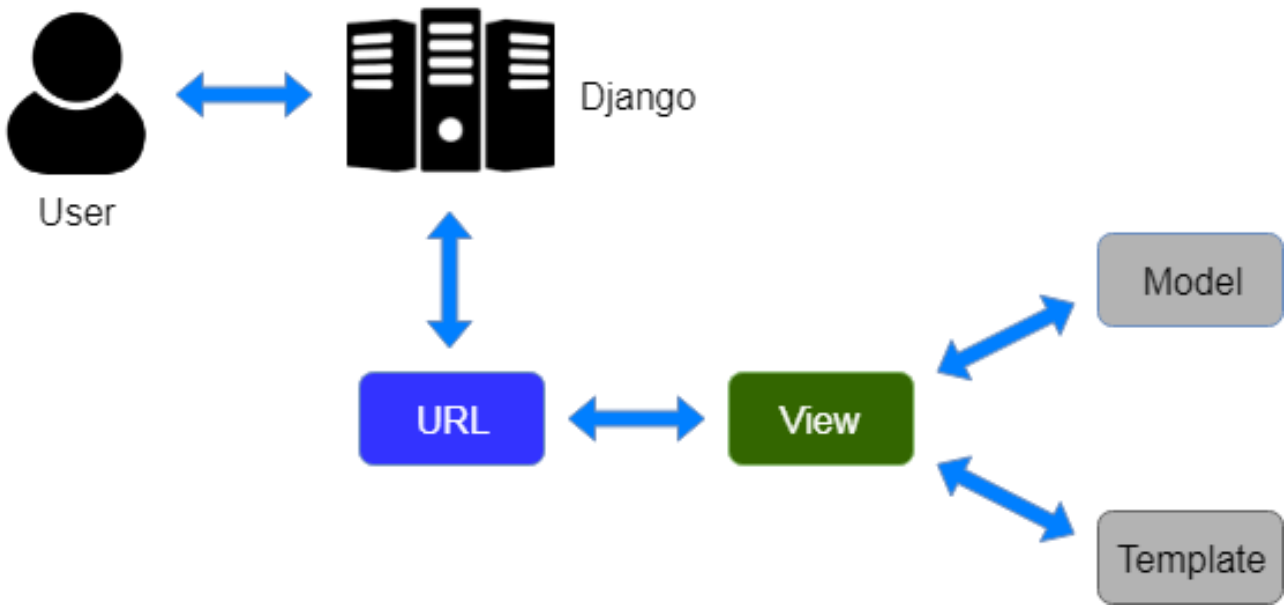
The MVT (Model View Template) is a software design pattern. It is a collection of three important components Model View and Template. The Model helps to handle database. It is a data access layer which handles the data.

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

Although Django follows MVC pattern but maintains it?s own conventions. So, control is handled 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, **a view is called** that interact with model and template, it renders a template.

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

Django Model

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

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

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

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

Let's see an example here, we are creating a model **Employee** which has two fields **first_name** and **last_name**.

```
from django.db import models

class Employee(models.Model):
    first_name = models.CharField(max_length=30)
    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.

```
CREATE TABLE appname_employee (
    "id" INT NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

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

Register / Use Model

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

For example,

```
INSTALLED_APPS = [
    #...
    'appname',
    #...
]
```

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,	It inherits all attributes and methods from FileField, but also

	height_field=None, width_field=None, max_length=100, **options)	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

```
first_name = models.CharField(max_length=50)      # for creating varchar column
release_date = models.DateField()                 # for creating date column
num_stars = models.IntegerField()                 # for creating integer column
```

Field Options

Each field requires some arguments that are used to set column attributes. For example, CharField requires max_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.

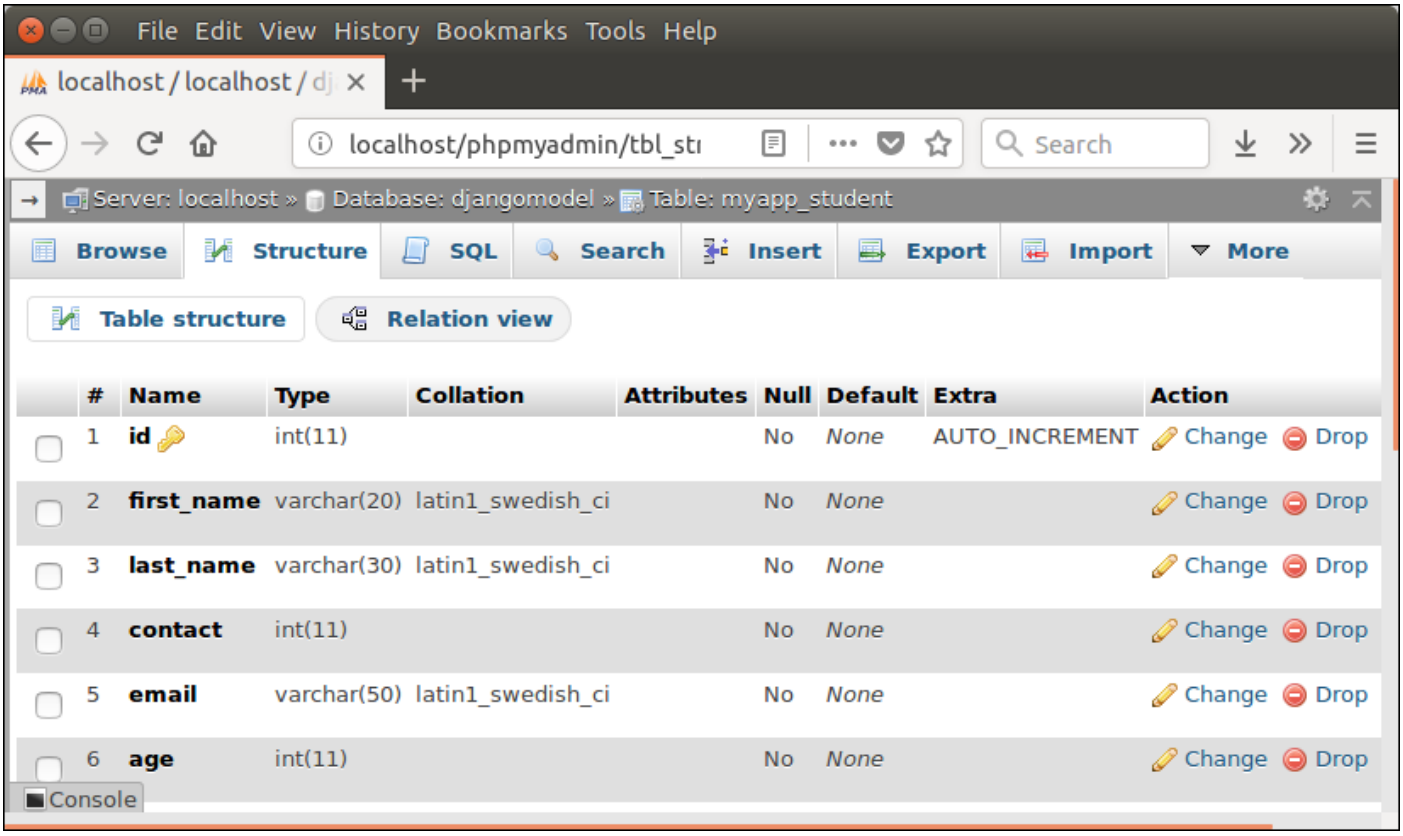
//models.py

```
class Student(models.Model):
    first_name = models.CharField(max_length=20)
    last_name  = models.CharField(max_length=30)
    contact    = models.IntegerField()
    email      = models.EmailField(max_length=50)
    age        = models.IntegerField()
```

After that apply migration by using the following command.

python3 manage.py makemigrations myapp

It will create a table **myapp_student**. The table structure looks like the below.



Django Views

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

All the view functions are created inside the **views.py** file of the Django app.

Django View Simple Example

//views.py

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

Let's step through the code.

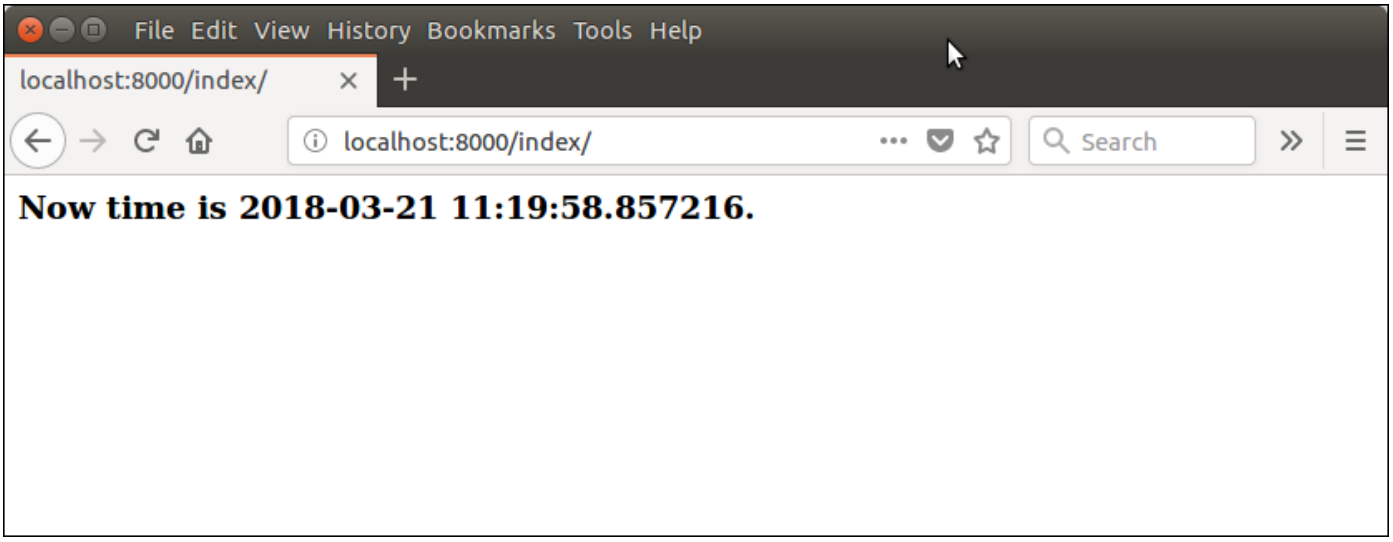
First, we will import DateTime library that provides a method to get current date and time and HttpResponse class.

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

```
path('index/', views.index),
```

Output:



Returning Errors

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

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

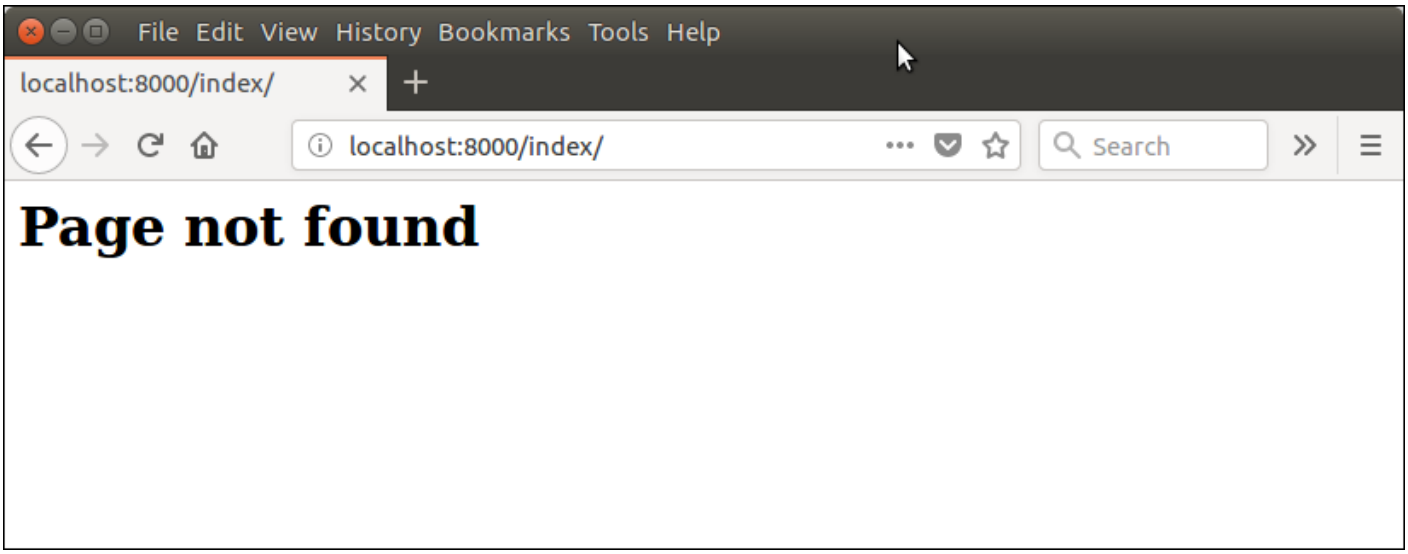
Django View Example

// views.py

```
from django.shortcuts import render
# Create your views here.
from django.http import HttpResponse, HttpResponseNotFound
def index(request):
    a = 1
    if a:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
```

```
    return HttpResponse('<h1>Page was found</h1>') # rendering the template in HttpResponse
```

Output:



Django View HTTP Decorators

HTTP Decorators are used to restrict 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

```
from django.shortcuts import render
# Create your views here.
from django.http import HttpResponse, HttpResponseNotFound
from django.views.decorators.http import require_http_methods
@require_http_methods(["GET"])
def show(request):
    return HttpResponse('<h1>This is Http GET request.</h1>')
```

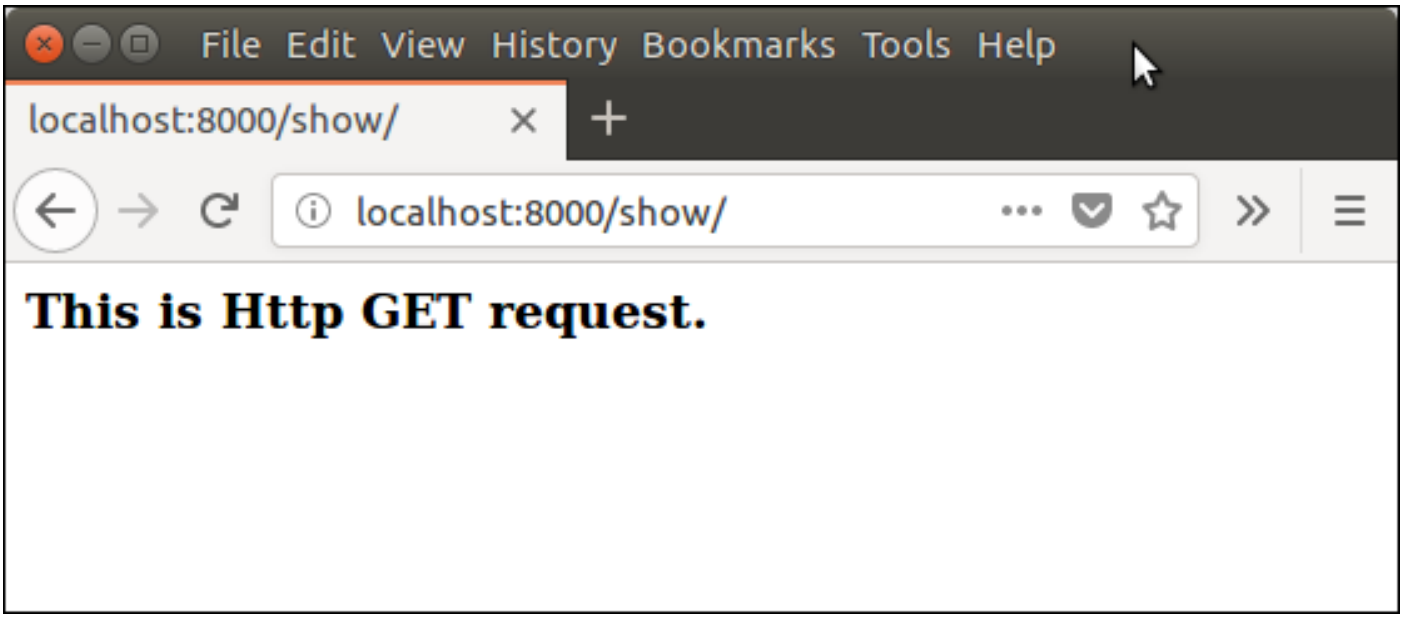
This method will execute only if the request is an HTTP GET request.

//urls.py

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
```

```
    path('show/',  views.show),
]
```

Output:



Django Templates

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

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

Why Django Template?

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

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

Django Template Configuration

To configure the template system, we have to provide some entries in **settings.py** file.

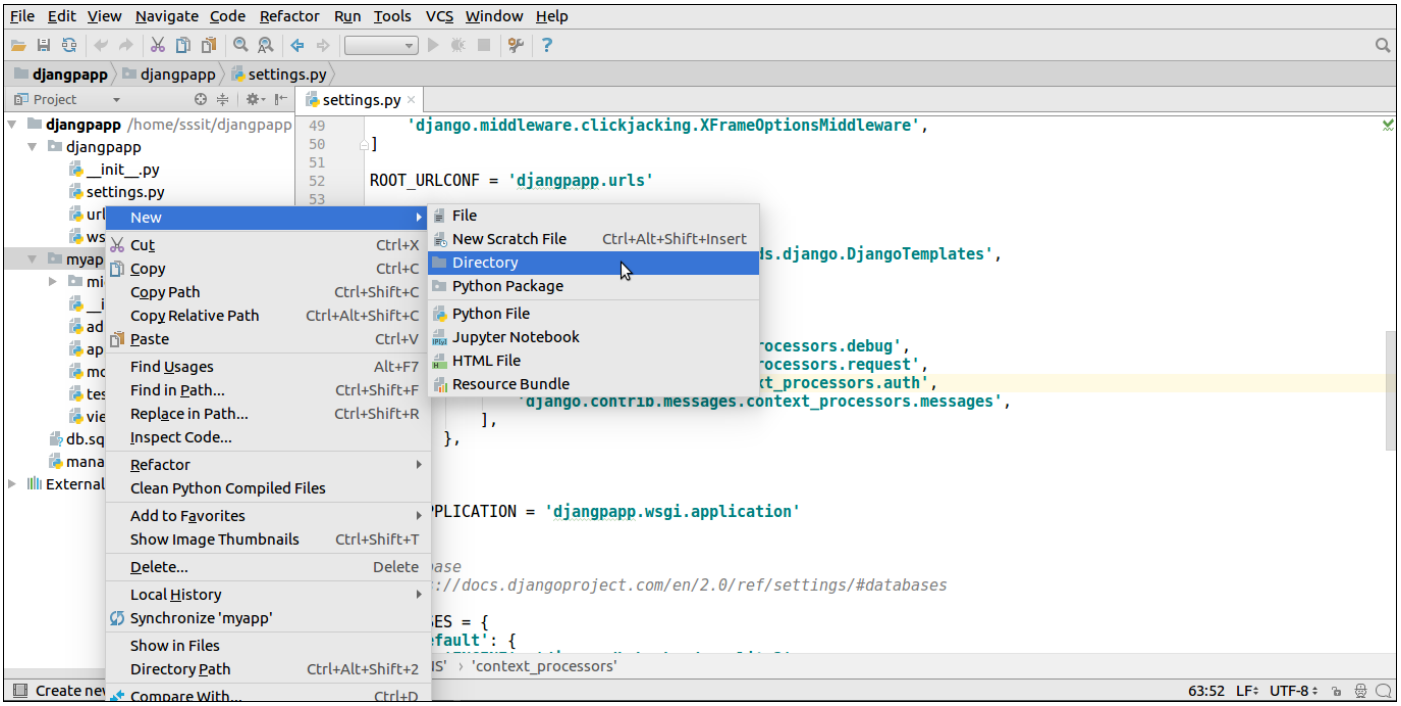
```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR,'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
```

```
            'django.contrib.messages.context_processors.messages',
        ],
    },
  },
]
```

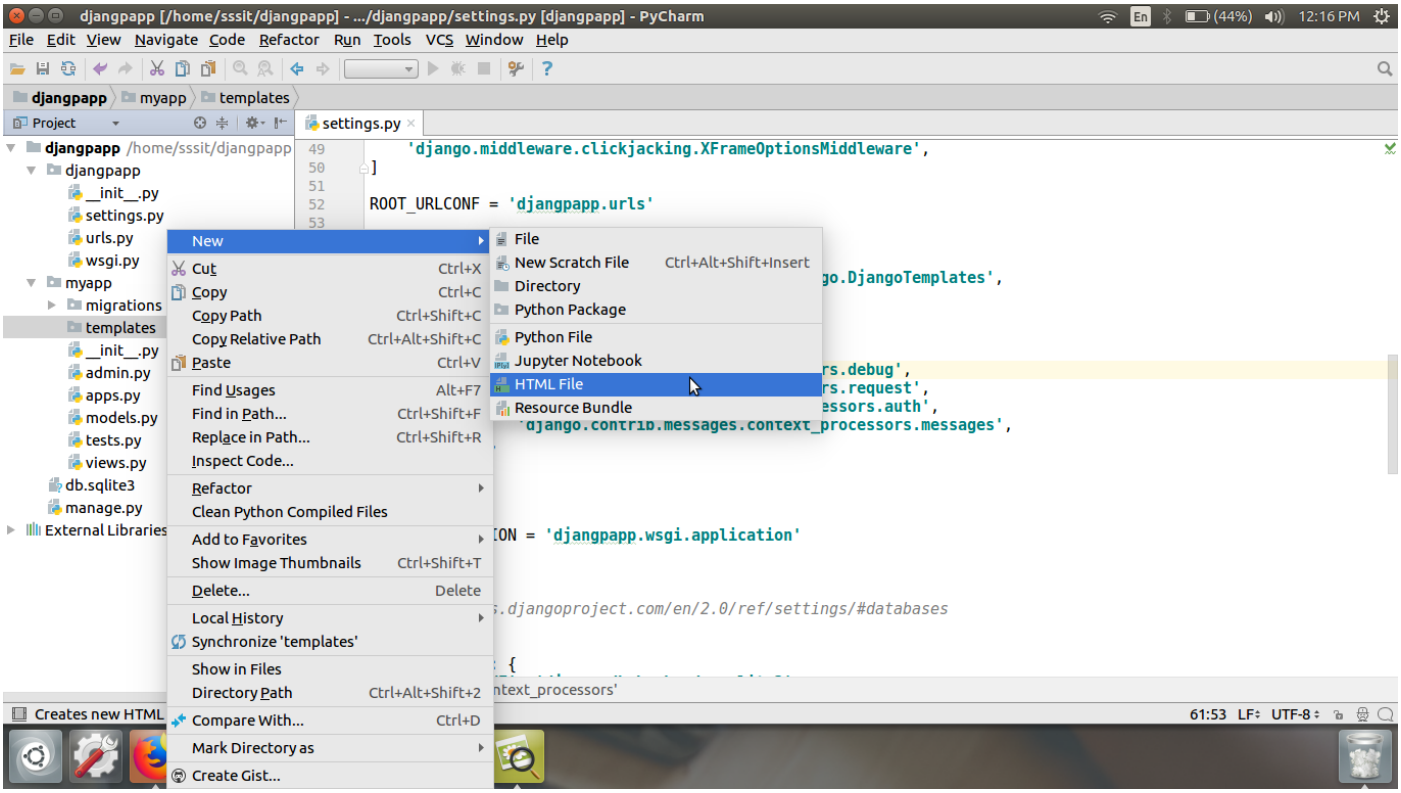
Here, we mentioned that our template directory name is **templates**. By default, DjangoTemplates looks for a **templates** subdirectory in each of the INSTALLED_APPS.

Django Template Simple Example

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



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



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

// index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
</head>
<body>
<h2>Welcome to Django!!!</h2>
</body>
</html>
```

Loading Template

To load the template, call get_template() method as we did below and pass template name.

//views.py

```
from django.shortcuts import render
#importing loading from django template
from django.template import loader
# Create your views here.
from django.http import HttpResponse
def index(request):
   template = loader.get_template('index.html') # getting our template
   return HttpResponse(template.render())       # rendering the template in HttpResponse
```

Set a URL to access the template from the browser.

//urls.py

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

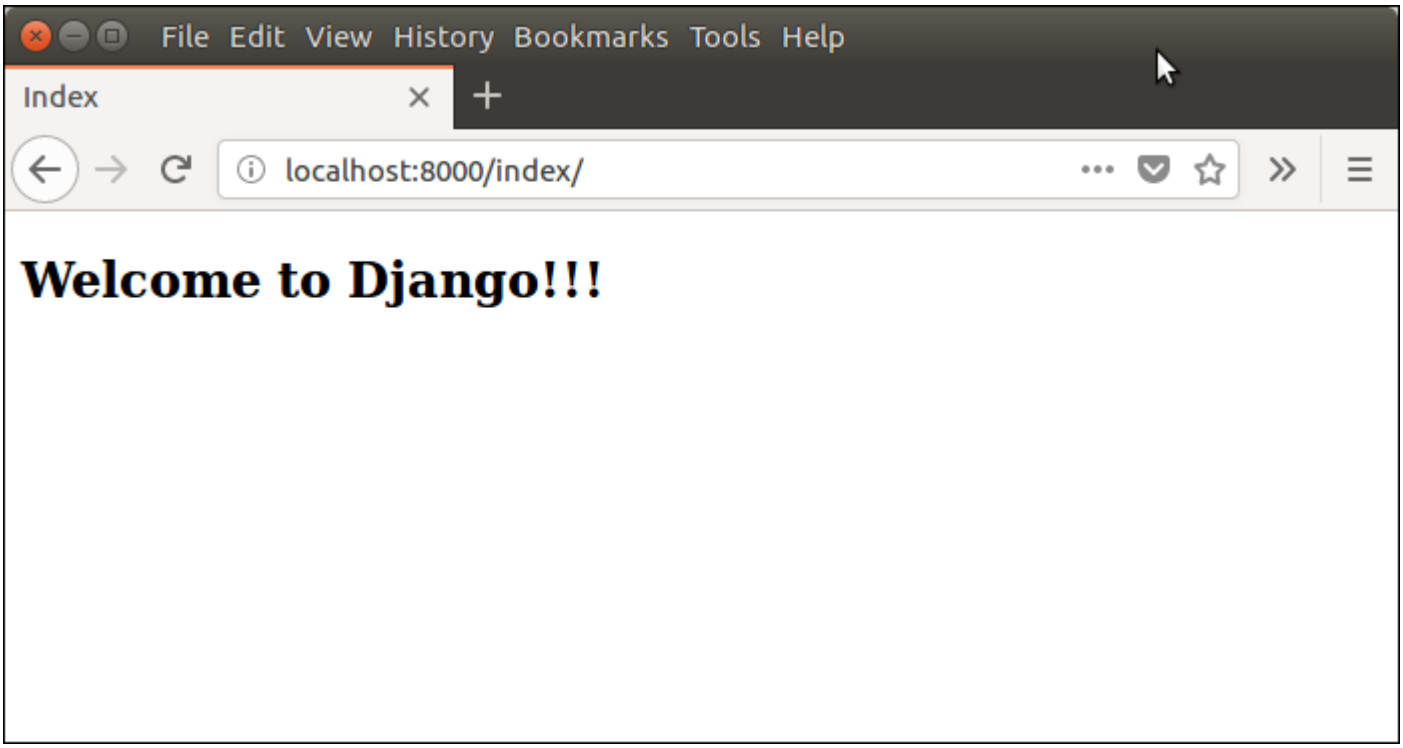
Register app inside the INSTALLED_APPS

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp'
]
```

Run Server

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

$ python3 manage.py runserver



Django Template Language

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

Variables

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

My name is {{name}}.
My name is rahul

Django Variable Example

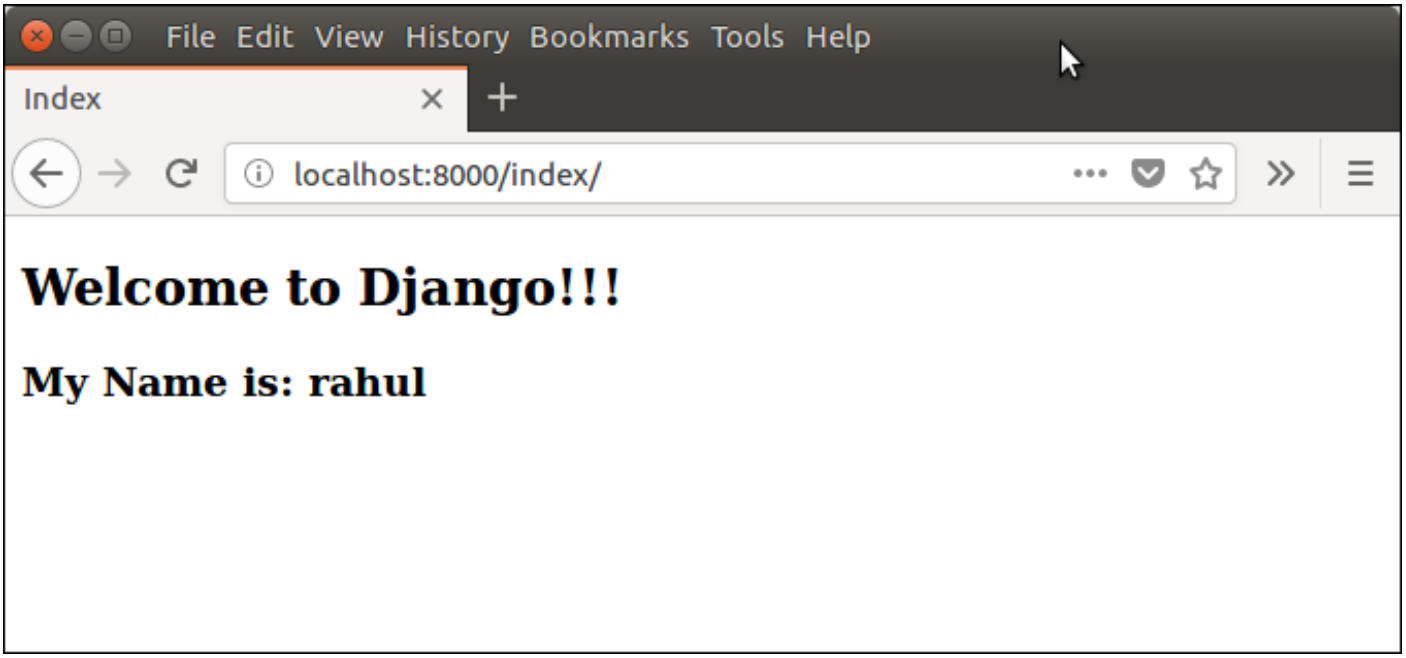
//views.py

```
from django.shortcuts import render
#importing loading from django template
from django.template import loader
# Create your views here.
from django.http import HttpResponse
def index(request):
    template = loader.get_template('index.html') # getting our template
    name = {
        'student':'rahul'
    }
    return HttpResponse(template.render(name))      # rendering the template in HttpResponse
```

//index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
</head>
<body>
<h2>Welcome to Django!!!</h2>
<h3>My Name is: {{ student }}</h3>
</body>
</html>
```

Output:



Tags

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

Tags are surrounded by {% %} braces. For example.

```
{% csrf_token %}

{% if user.is_authenticated %}
    Hello, {{ user.username }}.
{% endif %}
```

Django URL Mapping

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

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

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

// urls.py

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

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

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

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

The **django.urls** module contains various functions, **path(route,view,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.	

Let's see an example that gets user request and map that route to call specified view function. Have a look at the steps.

Create a function **hello** in the views.py file. This function will be mapped from the url.py file.

// views.py

```
from django.shortcuts import render
# Create your views here.
from django.http import HttpResponse, HttpResponseNotFound
from django.views.decorators.http import require_http_methods
@require_http_methods(["GET"])
def hello(request):
    return HttpResponse('<h1>This is Http GET request.</h1>')
```

// urls.py

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

```
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
    path('hello/',  views.hello),
]
```

Now, start the server and enter **localhost:8000/hello** to the browser. This URL will be mapped into the list of URLs and then call the corresponding function from the views file.

In this example, hello will be mapped and call hello function from the views file. It is called URL mapping.

Django Static Files Handling

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

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

Django deals with it 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.**

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp'
]
```

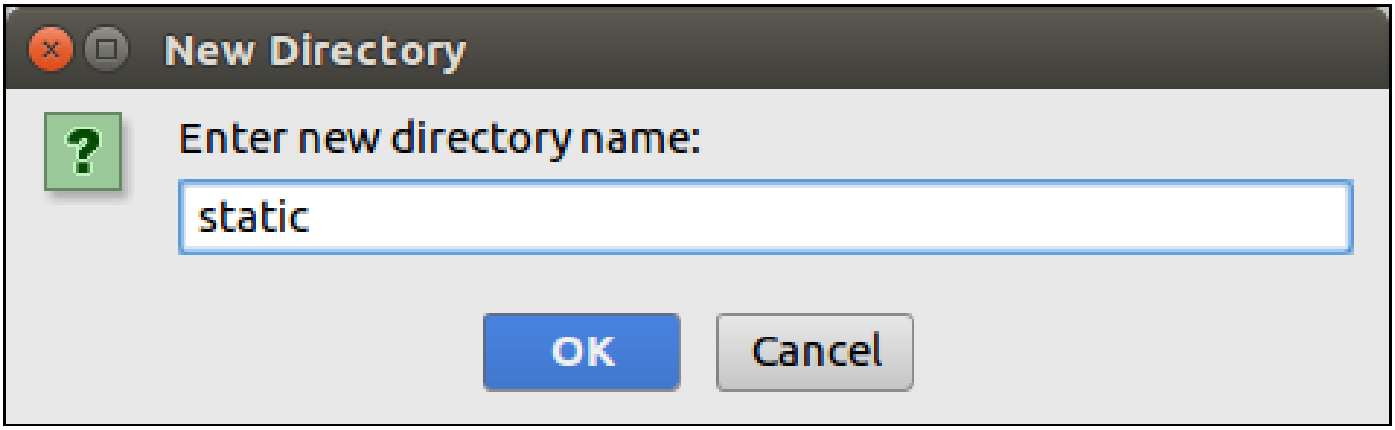
2. Define STATIC_URL in settings.py file as given below.

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

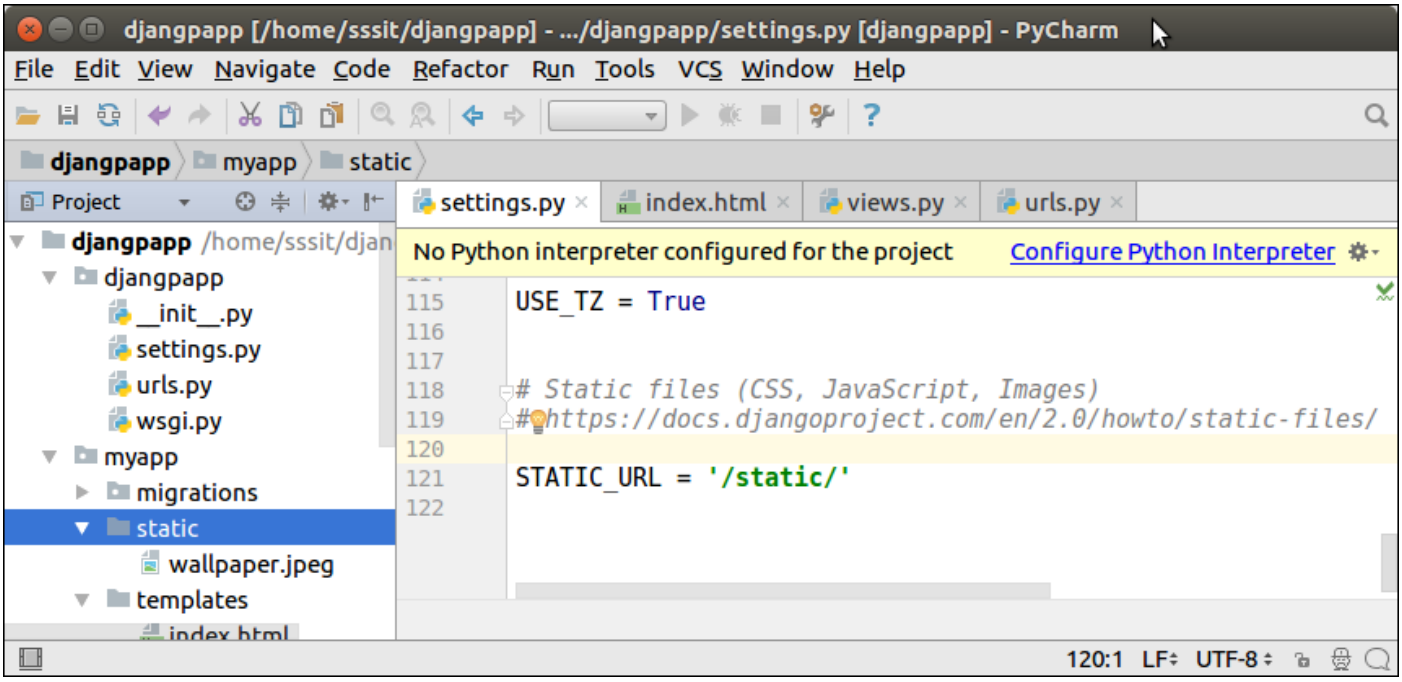
3. Load static files in the templates by using the below expression.

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

4. Store all images, JavaScript, CSS files in a **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 an image in a template file, use the code given below.

// index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
     {% load static %}
</head>
<body>
<img src="{% static '/wallpaper.jpeg' %}" alt="My image" height="300px" width="700px"/>
</body>
</html>
```

//urls.py

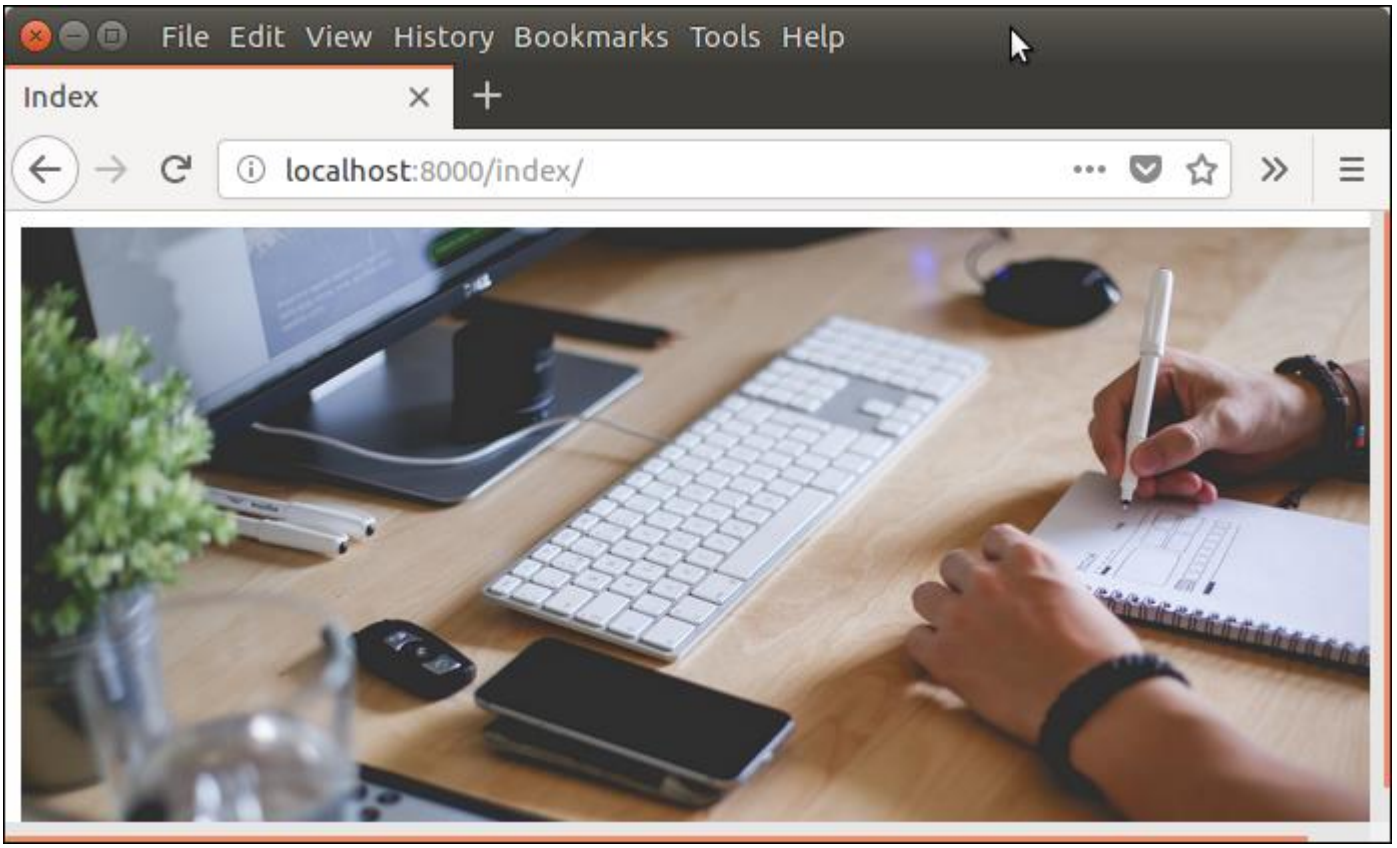
```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
]
```

//views.py

```
def index(request):
    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.

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

// index.html

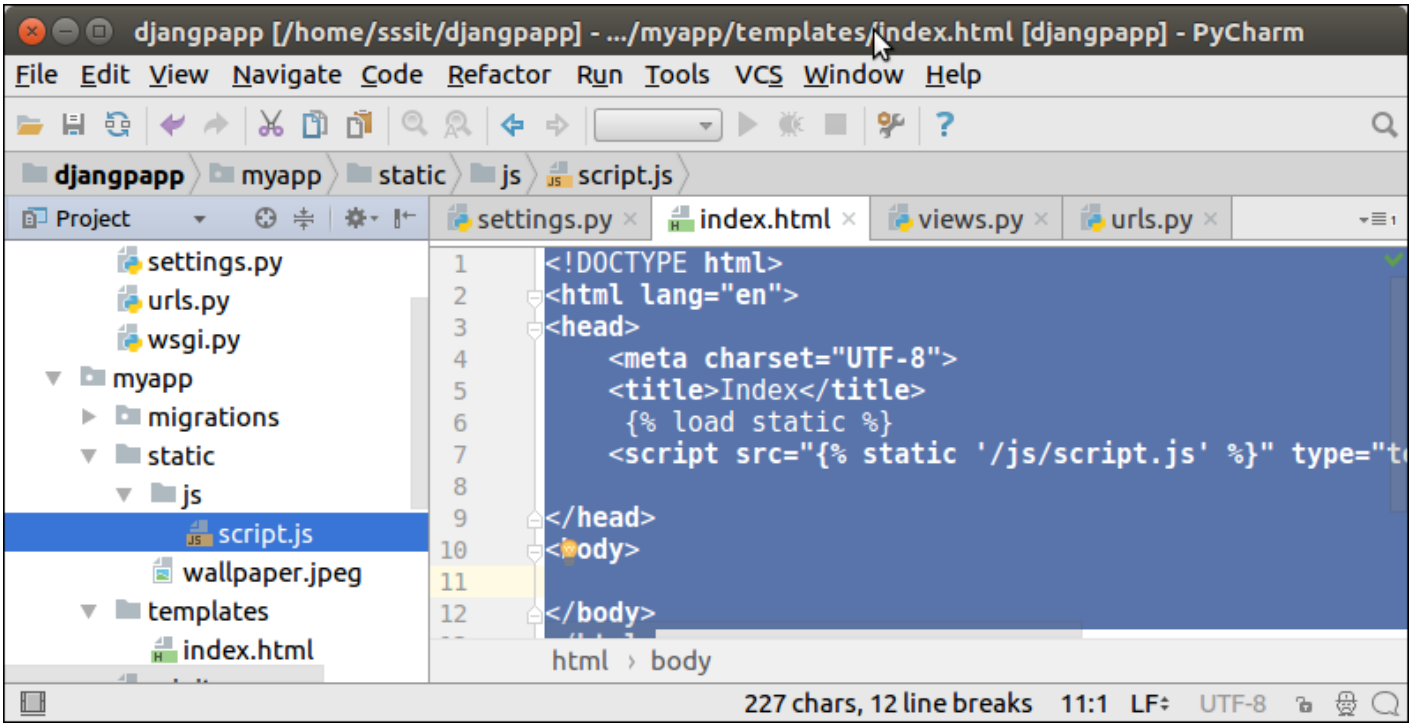
```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
     {% load static %}
    <script src="{% static '/js/script.js' %}" type="text/javascript"></script>
</head>
<body>
</body>
</html>
```

// script.js

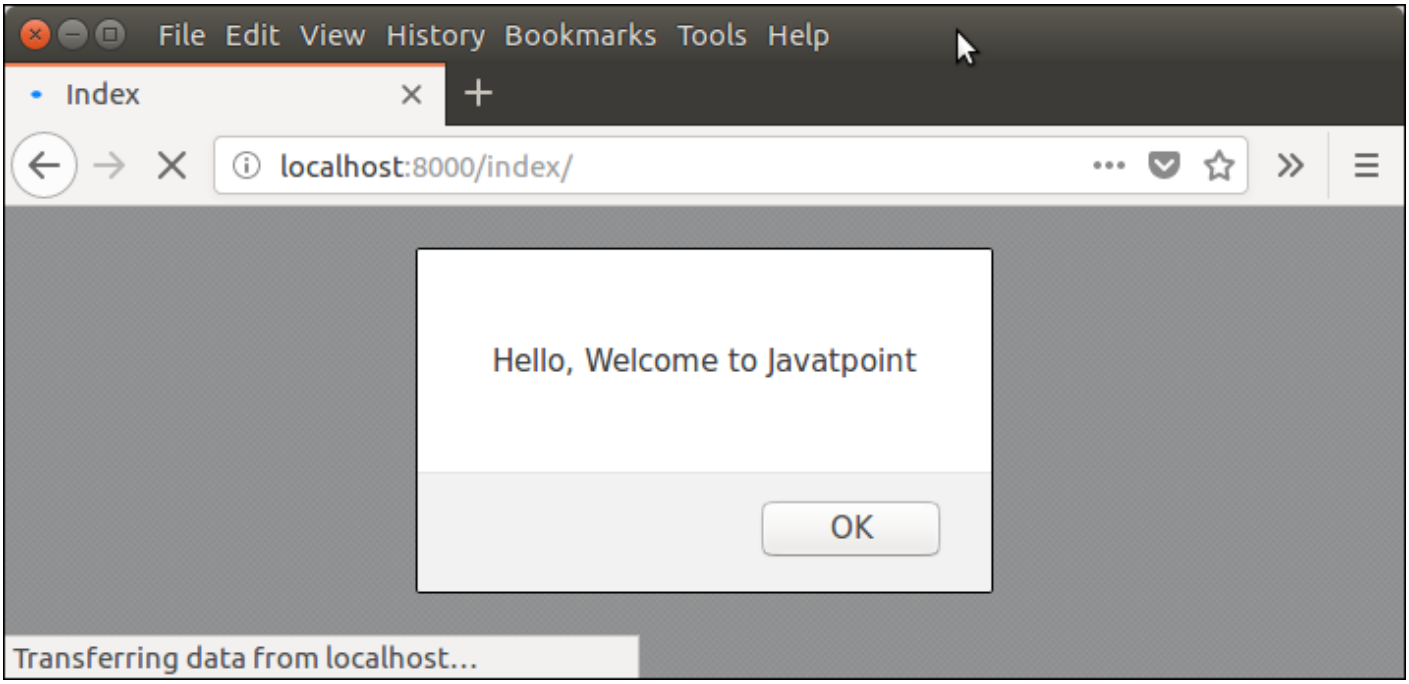
```
alert("Hello, Welcome to Javatpoint");
```

Now, our project structure looks like 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.

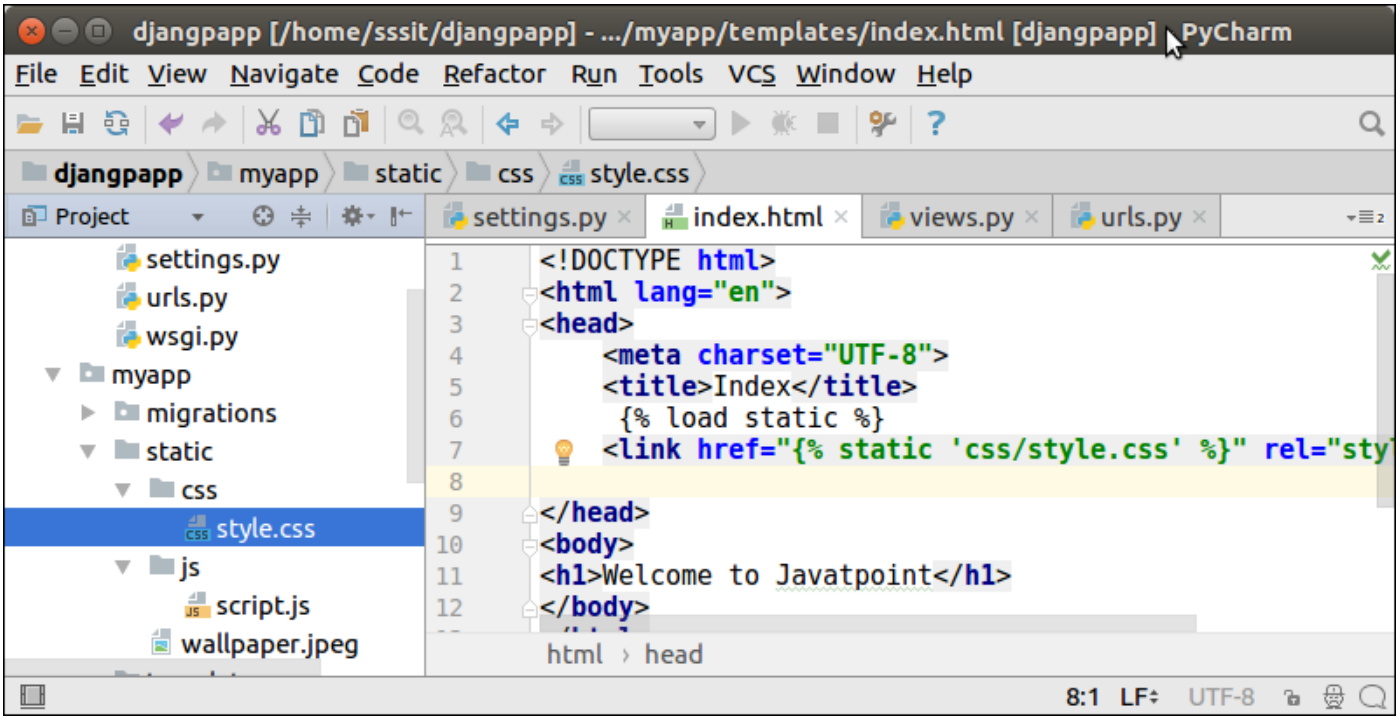
```
{% load static %}
<link href="{% static 'css/style.css' %}" rel="stylesheet">
```

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

// style.css

```
h1{
color:red;
}
```

Our project structure looks like this:

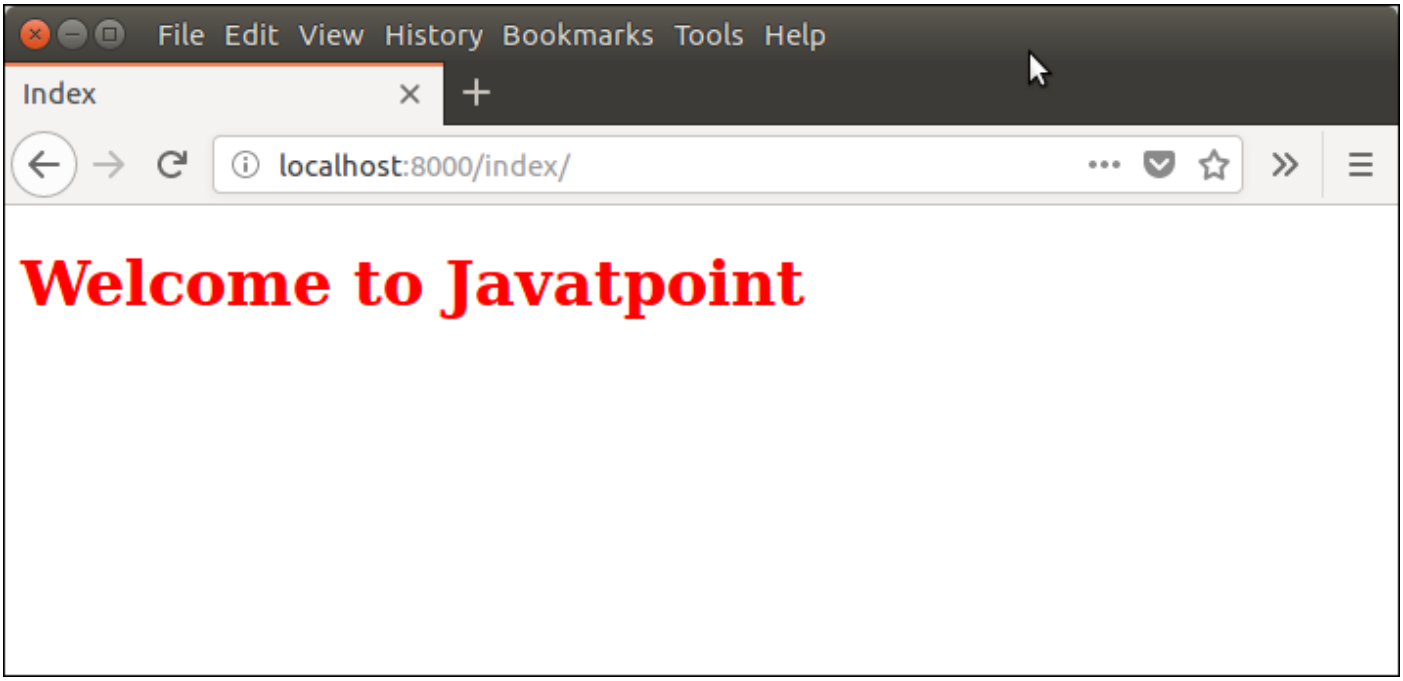


// index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
     {% load static %}
    <link href="{% static 'css/style.css' %}" rel="stylesheet">
</head>
<body>
<h1>Welcome to Javatpoint</h1>
</body>
</html>
```

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

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



Well, in this topic, we have learned the process of managing static files efficiently.

Django Model Form

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

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

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

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 a table in database and dynamic HTML form.

// model.py

```
from __future__ import unicode_literals
from django.db import models

class Student(models.Model):
    first_name = models.CharField(max_length=20)
    last_name  = models.CharField(max_length=30)
    class Meta:
        db_table = "student"
```

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

// form.py

```
from django import forms
from myapp.models import Student

class EmpForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = "__all__"
```

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

//views.py

```
from django.shortcuts import render
from myapp.form import StuForm

def index(request):
    stu = StuForm()
    return render(request,"index.html",{'form':stu})
```

//urls.py

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
]
```

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

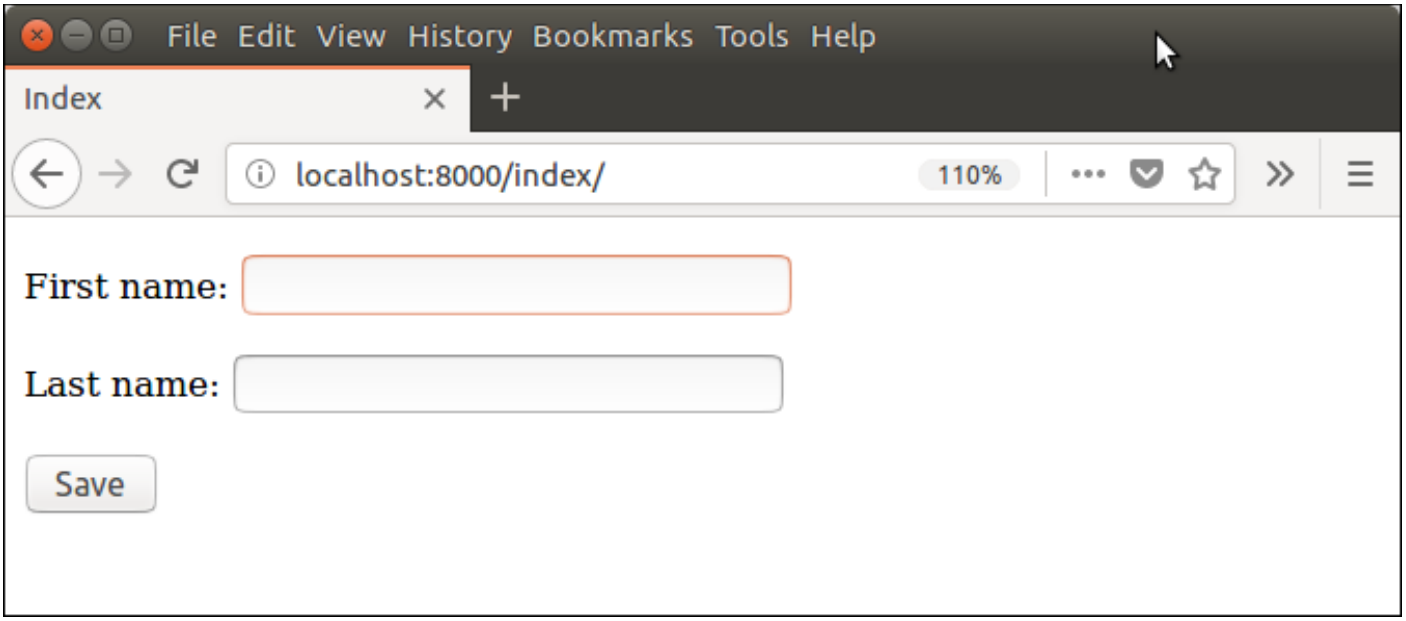
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
</head>
<body>
<form method="POST" class="post-form">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
    </form>
</body>
</html>
```

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 form is created automatically. This is a feature of Django.

Django Forms

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

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

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

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

```
from django import forms
class StudentForm(forms.Form):
    firstname = forms.CharField(label="Enter first name",max_length=50)
    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 an HTML text input component in the form.

The label is used to set HTML label of the component and max_length sets length of an input value.

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

```
<label for="id_firstname">Enter first name:</label>
 <input type="text" name="firstname" required maxlength="50" id="id_firstname" />
<label for="id_lastname">Enter last name:</label> <input type="text" name="lastname" required m
axlength="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

Let's 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.

```
from django import forms
class StudentForm(forms.Form):
    firstname = forms.CharField(label="Enter first name",max_length=50)
```

```
    lastname  = forms.CharField(label="Enter last name", max_length = 100)
```

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

Instantiating Form in Django

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

// views.py

```
from django.shortcuts import render
from myapp.form import StudentForm

def index(request):
    student = StudentForm()
    return render(request,"index.html",{'form':student})
```

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

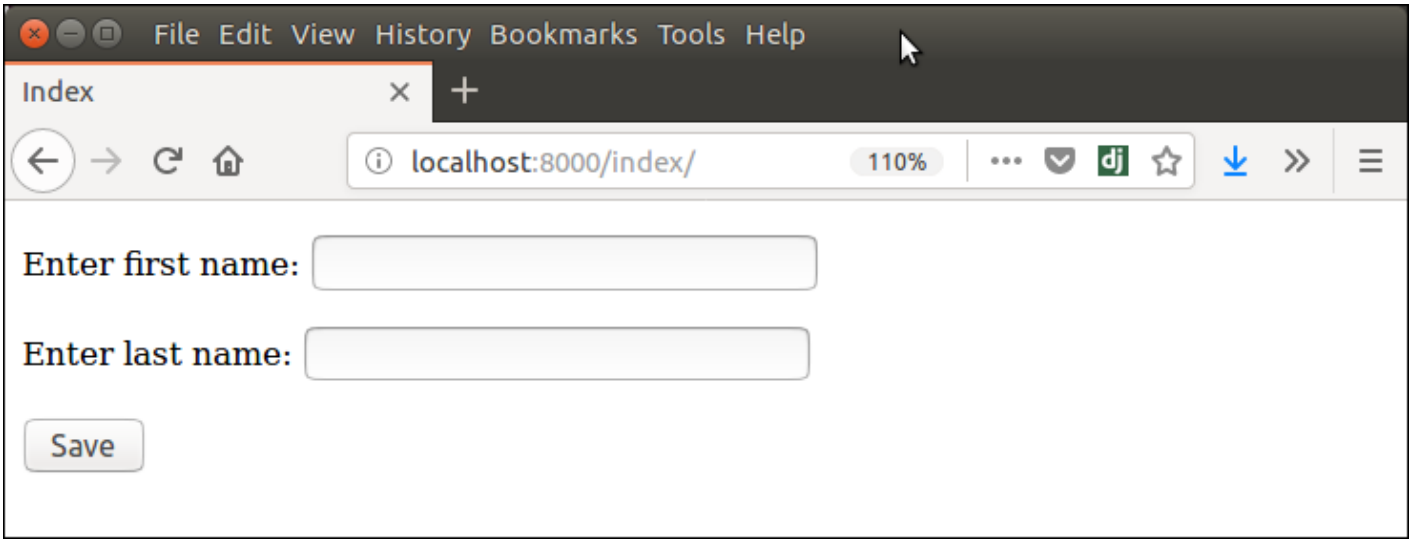
// index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
</head>
<body>
<form method="POST" class="post-form">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
</form>
</body>
</html>
```

Provide the URL in urls.py

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
]
```

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 we'll have to provide the surrounding <table> or <ul> elements yourself.*

Django Form Validation

Django provides built-in methods to validate form data automatically. Django forms submit only if it contains CSRF tokens. It uses uses a clean and easy approach to validate data.

The **is_valid()** method is used to perform validation for each field of the form, it is defined in Django Form class. It returns True if data is valid and place all data into a cleaned_data attribute.

Let's see an example that takes user input and validate input as well.

Django Validation Example

This example contains the following files and code.

// models.py

```
from django.db import models
class Employee(models.Model):
    eid = models.CharField(max_length=20)
    ename = models.CharField(max_length=100)
    econtact = models.CharField(max_length=15)
    class Meta:
        db_table = "employee"
```

Now, create a form which contains the below code.

// forms.py

```
from django import forms
```

```
from myapp.models import Employee

class EmployeeForm(forms.ModelForm):
    class Meta:
        model = Employee
        fields = "__all__"
```

Instantiate the form

Instantiate the form, check whether request is post or not. It validate the data by using **is_valid()** method.

//views.py

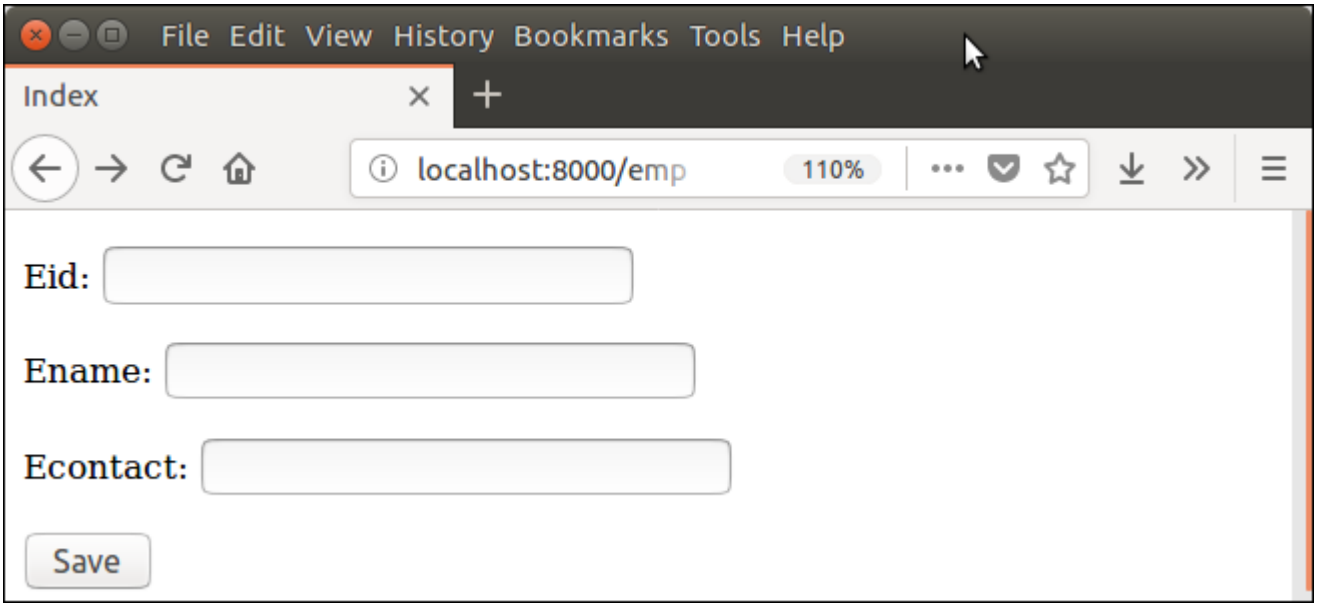
```
def emp(request):
    if request.method == "POST":
        form = EmployeeForm(request.POST)
        if form.is_valid():
            try:
                return redirect('/')
            except:
                pass
    else:
        form = EmployeeForm()
    return render(request,'index.html',{'form':form})
```

Index template that shows form and errors.

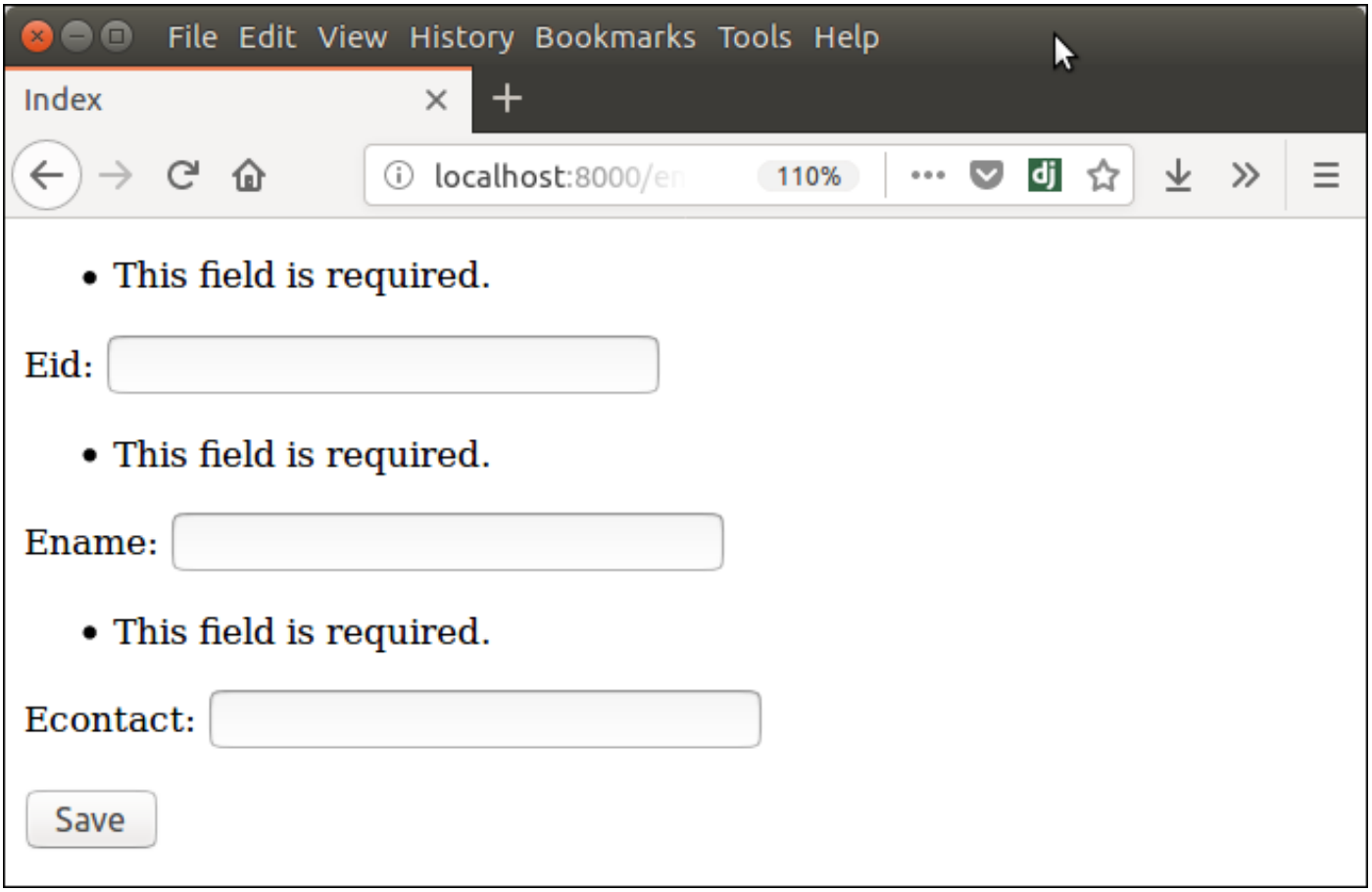
// index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
</head>
<body>
<form method="POST" class="post-form" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
</form>
</body>
</html>
```

Start server and access the form.



It validates each field and throws errors if any validation fails.



Django File Upload

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

The **forms.FileField()** method is used to create a file input and submit the file to the server. While working with files, make sure the HTML form tag contains **enctype="multipart/form-data"** property.

Let's see an example of uploading a file to the server. This example contains the following files.

Template (index.html)

It will create an HTML form which contains a file input component.

```
<body>
<form method="POST" class="post-form" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
</form>
</body>
```

Form (forms.py)

```
from django import forms
class StudentForm(forms.Form):
    firstname = forms.CharField(label="Enter first name",max_length=50)
    lastname  = forms.CharField(label="Enter last name", max_length = 10)
    email     = forms.EmailField(label="Enter Email")
    file      = forms.FileField() # for creating file input
```

View (views.py)

Here, one extra parameter **request.FILES** is required in the constructor. This argument contains the uploaded file instance.

```
from django.shortcuts import render
from django.http import HttpResponse
from myapp.functions.functions import handle_uploaded_file
from myapp.form import StudentForm
def index(request):
    if request.method == 'POST':
        student = StudentForm(request.POST, request.FILES)
        if student.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponse("File uploaded successfuly")
    else:
        student = StudentForm()
        return render(request,"index.html",{'form':student})
```

Specify URL (urls.py)

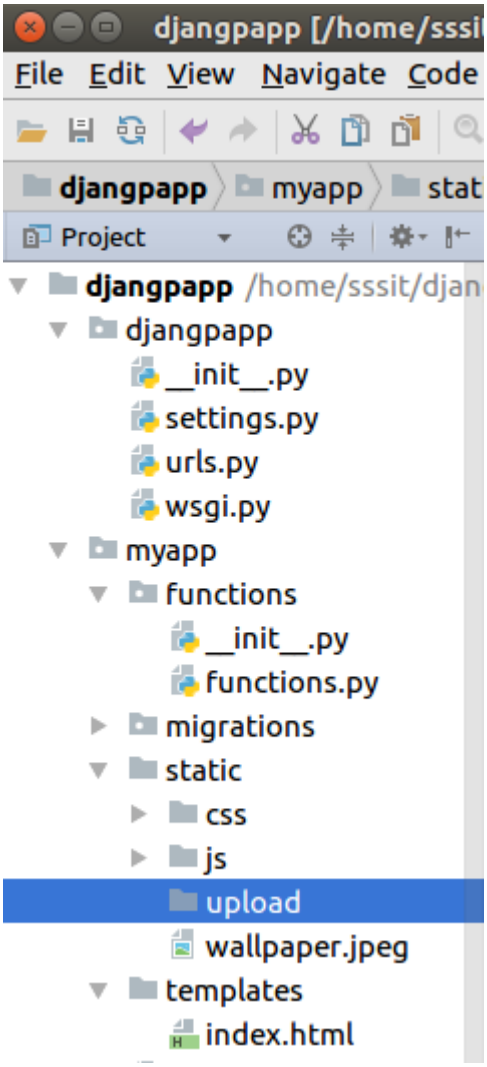
```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
]
```

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.

```
def handle_uploaded_file(f):
    with open('myapp/static/upload/'+f.name, 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

Now, create a directory **upload** to store the uploaded file. Our project structure looks like below.

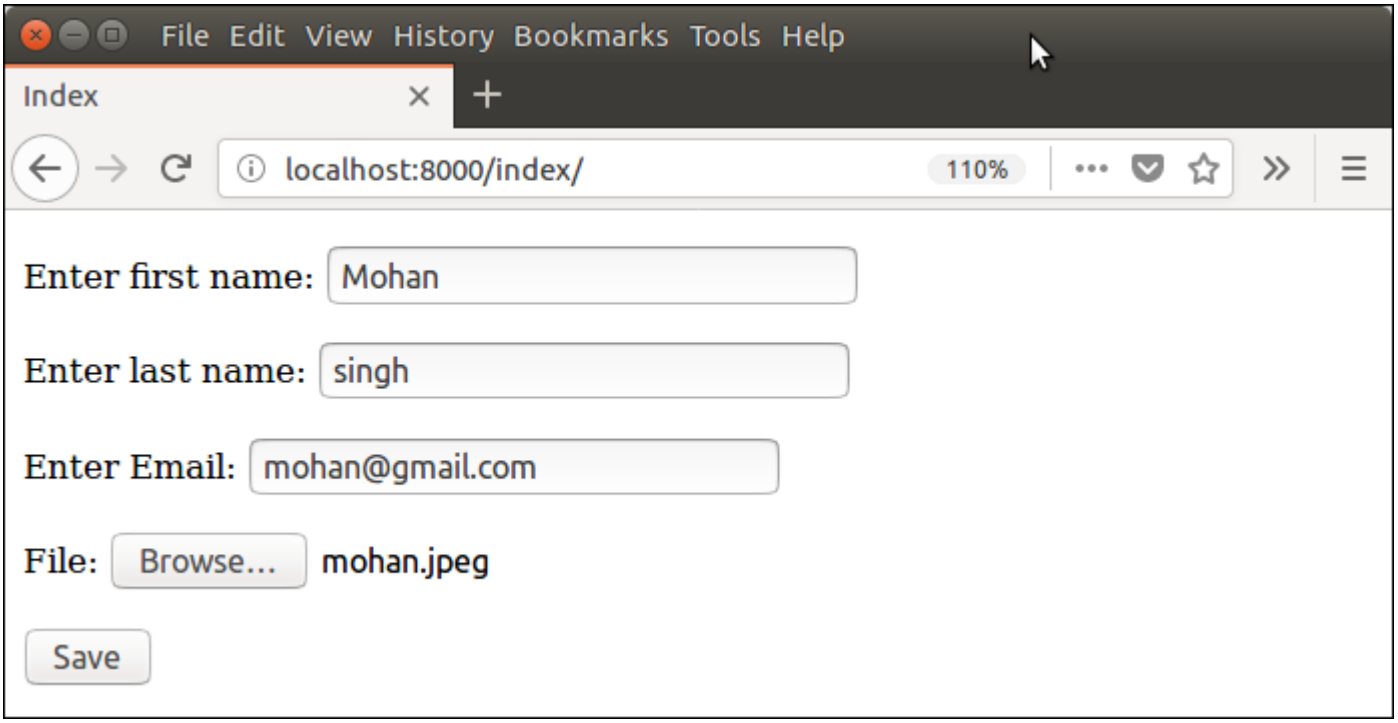


Initially, this directory is empty. so, let's upload a file to it and later on it will contain the uploaded file.

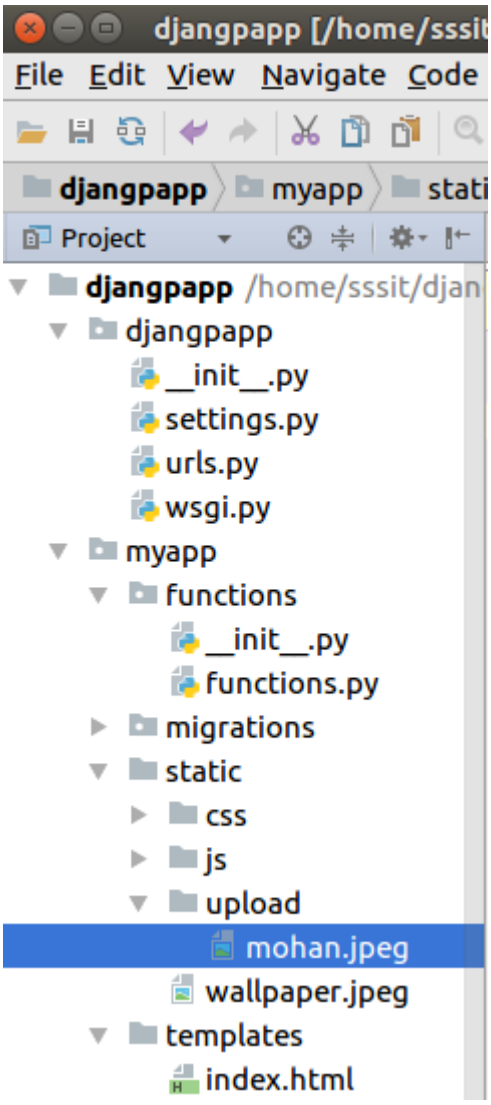
Start Server

```
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,** database and allows configuring for other databases as well.

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

To connect with MySQL, **django.db.backends.mysql** driver is used to establishing a connection between application and database. Let's see an example.

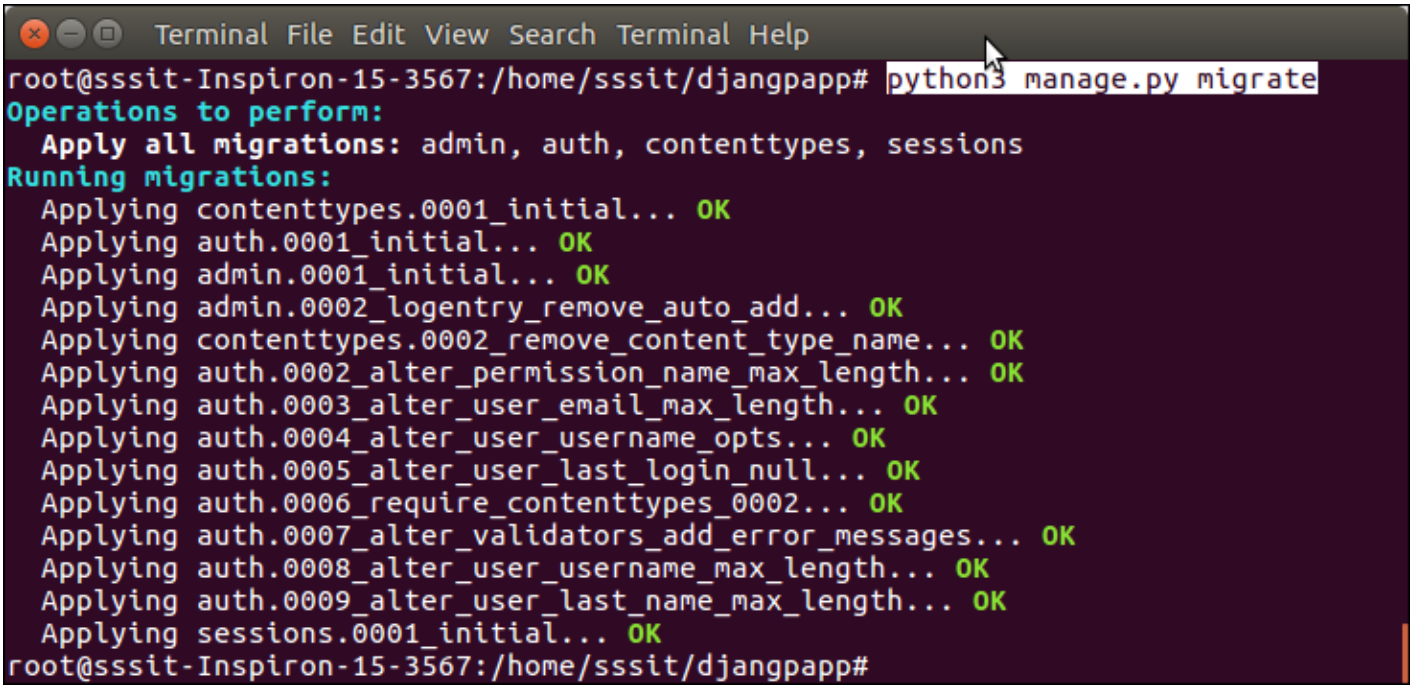
We need to provide all connection details in the settings file. The settings.py file of our project contains the following code for the database.

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

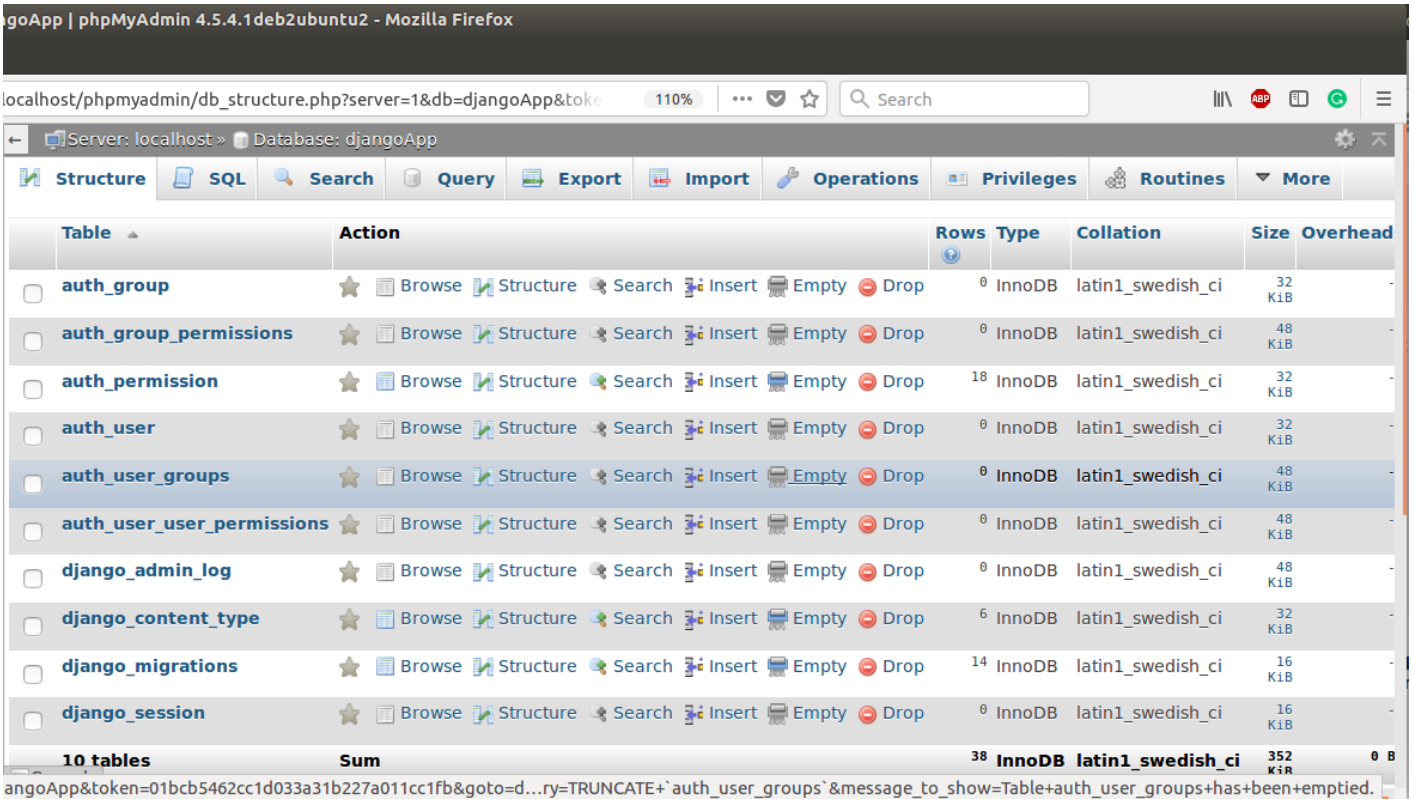
After providing details, check the connection using the migrate command.

```
1. $ python3 manage.py migrate
```

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



Now, access to the MySQL database and see the database from the list of databases. The created database contains the following tables.



> ***Note:*** *It throws an 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 learned to connect Django application to the MySQL database. Next, we will see how to create a table using the model.

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

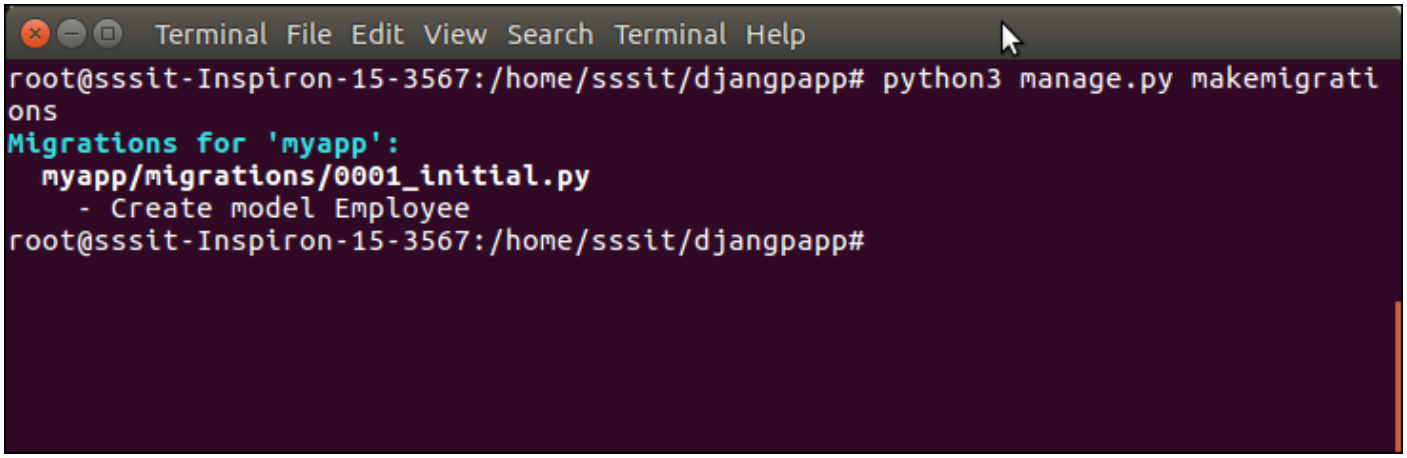
Suppose, we have a model class Employee in the **models.py** file that contains the following code.

// models.py

```
from django.db import models
class Employee(models.Model):
    eid      = models.CharField(max_length=20)
    ename    = models.CharField(max_length=100)
    econtact = models.CharField(max_length=15)
    class Meta:
        db_table = "employee"
```

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

```
$ python3 manage.py makemigrations
```

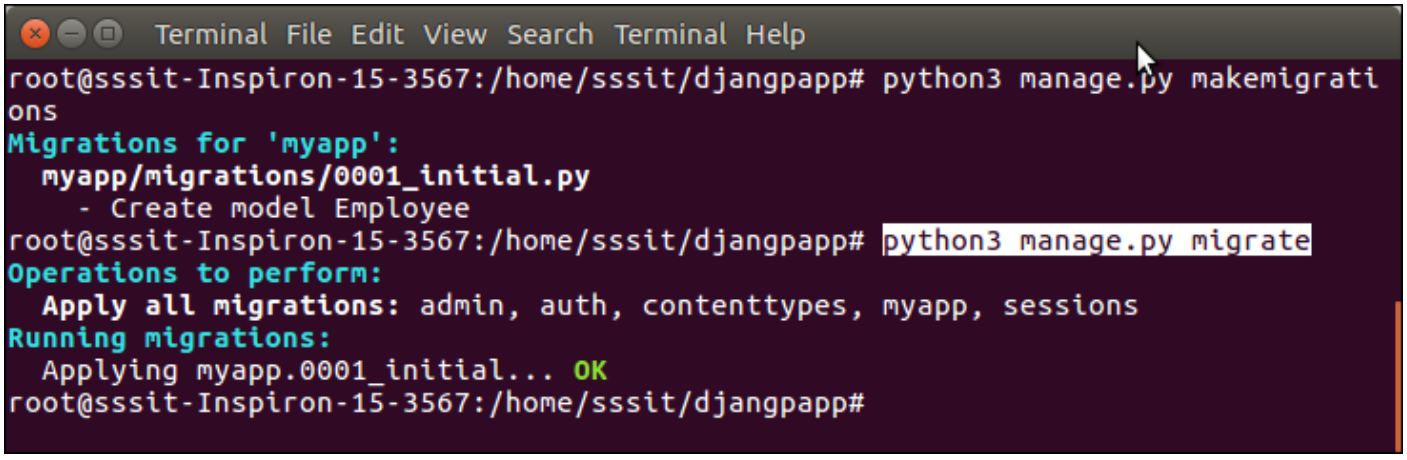


The created migration file is located into **migrations** folder and contains the following code.

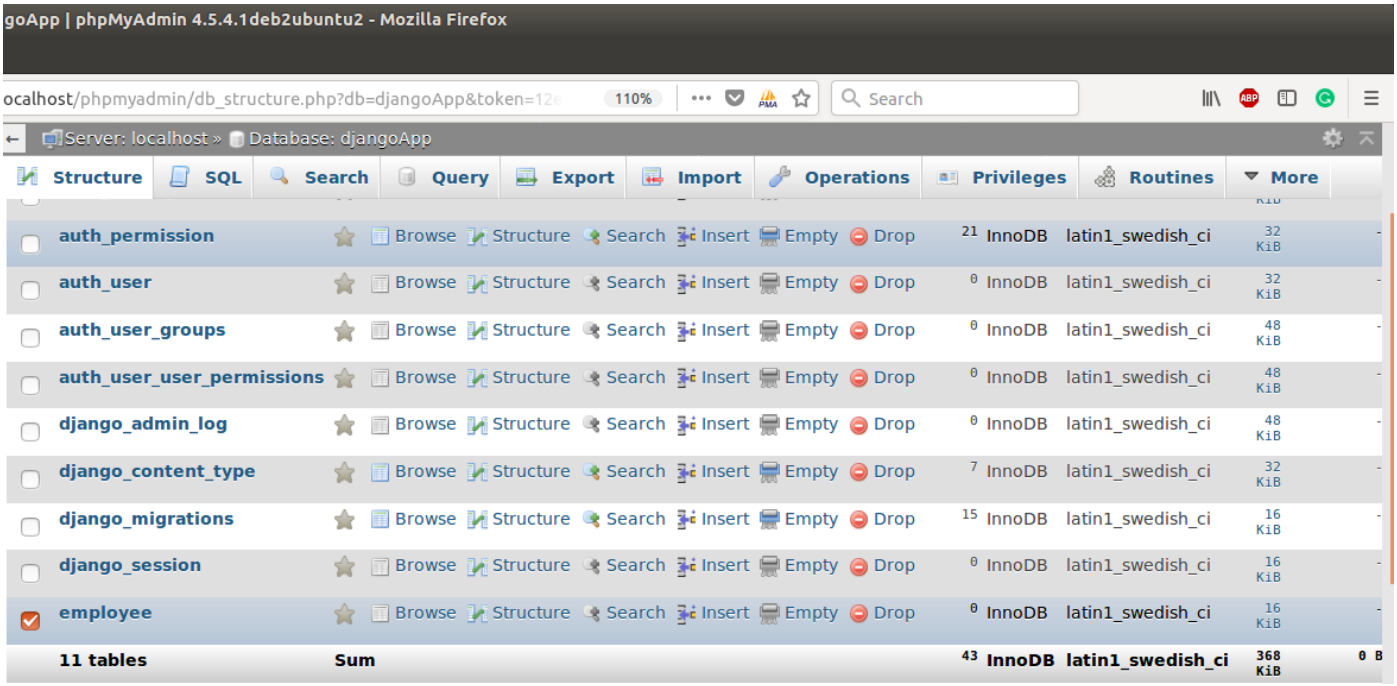
```
from django.db import migrations, models
class Migration(migrations.Migration):
    initial = True
    dependencies = [
    ]
    operations = [
        migrations.CreateModel(
            name='Employee',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_na
me='ID')),
                ('eid', models.CharField(max_length=20)),
                ('ename', models.CharField(max_length=100)),
                ('econtact', models.CharField(max_length=15)),
            ],
            options={
                'db_table': 'employee',
            },
        ),
    ]
```

Now, migrate to reflect the changes into the database.

1. **$ python3 manage.py migrate**
2.



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



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

Django Database Migrations

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

Django provides the various commands that are used to perform migration related tasks. After creating a model, we can use these commands.

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

Suppose, we have a model as given below and contains the following attributes.

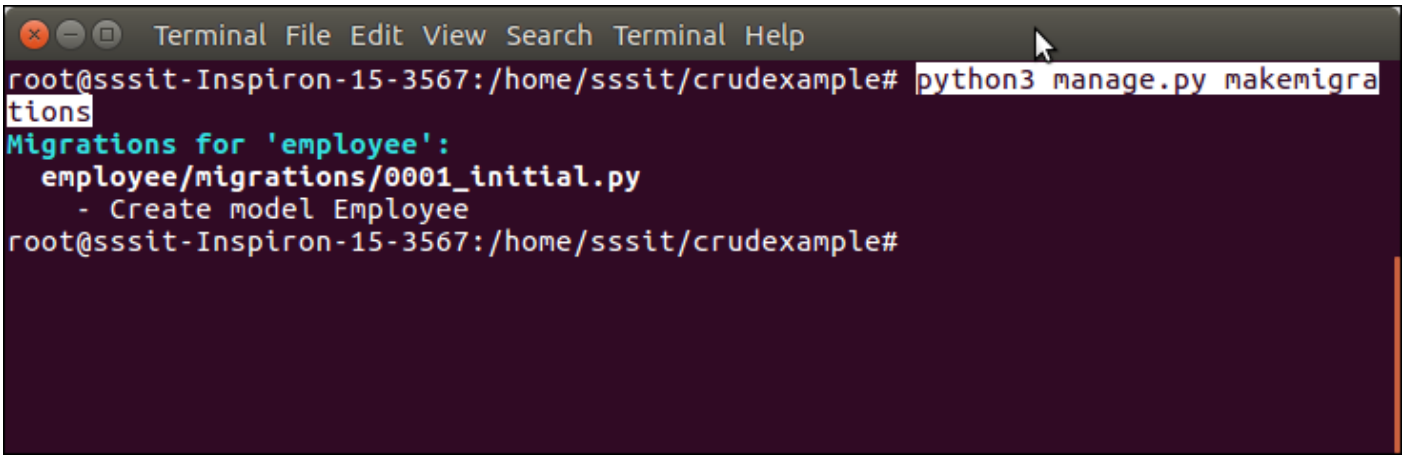
Model

//models.py

```
from django.db import models
class Employee(models.Model):
    eid = models.CharField(max_length=20)
    ename = models.CharField(max_length=100)
    econtact = models.CharField(max_length=15)
    class Meta:
        db_table = "employee"
```

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

$ 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.

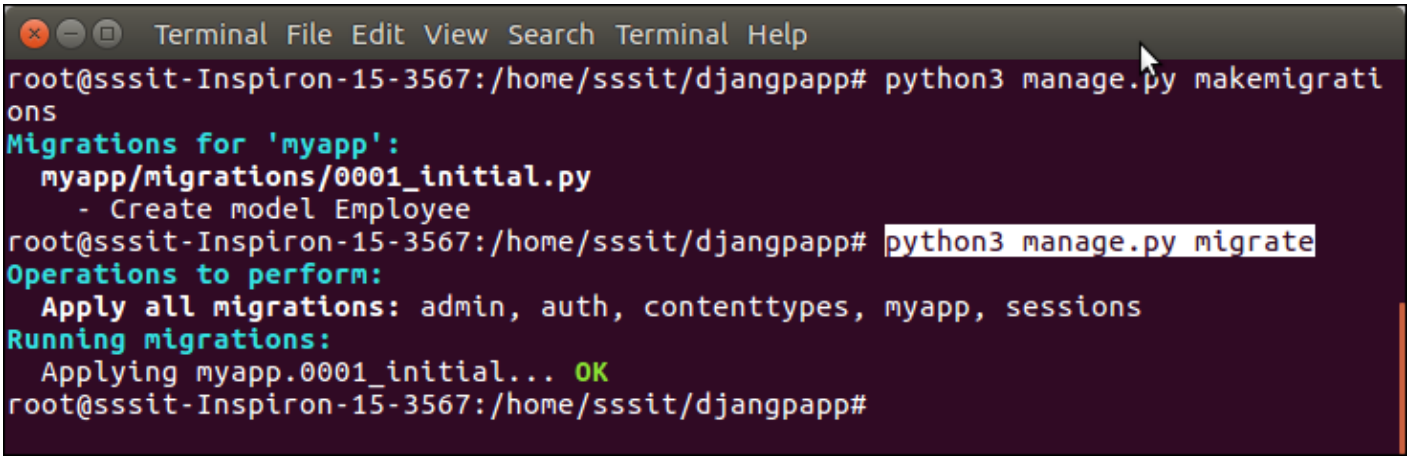
Migrations

// 0001_initial.py

```
from django.db import migrations, models
class Migration(migrations.Migration):
    initial = True
    dependencies = [
    ]
    operations = [
        migrations.CreateModel(
            name='Employee',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_na
me='ID')),
                ('eid', models.CharField(max_length=20)),
                ('ename', models.CharField(max_length=100)),
                ('econtact', models.CharField(max_length=15)),
            ],
            options={
                'db_table': 'employee',
            },
        ),
    ]
```

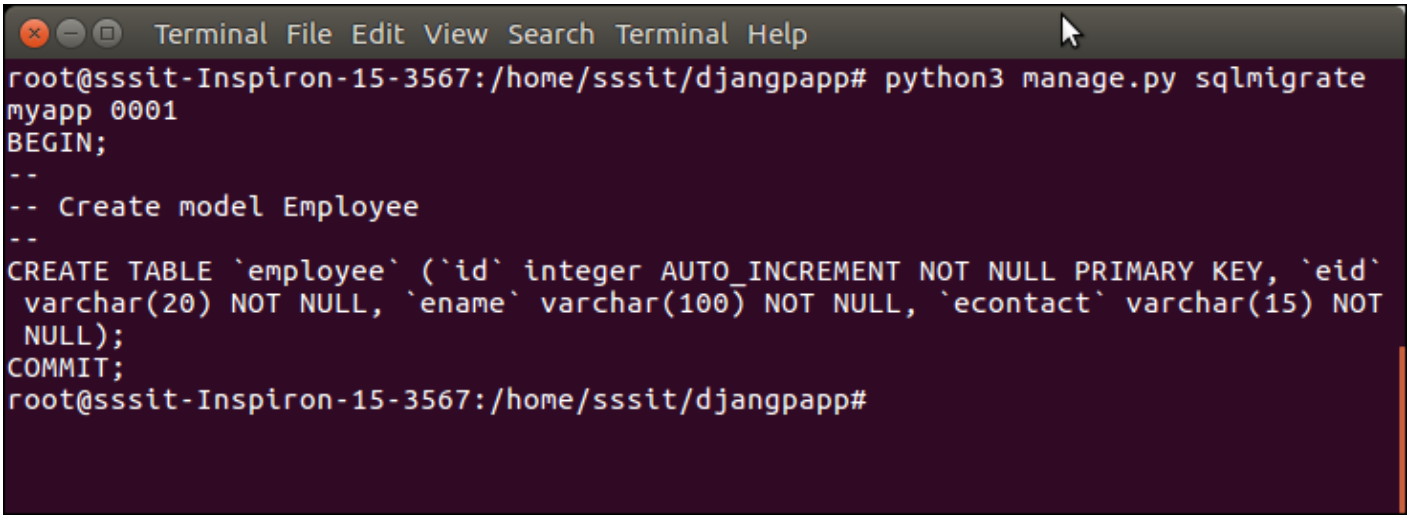
After creating a migration, migrate it so that it reflects the database permanently. The migrate command is given below.

$ python3 manage.py migrate



Apart from creating a 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.

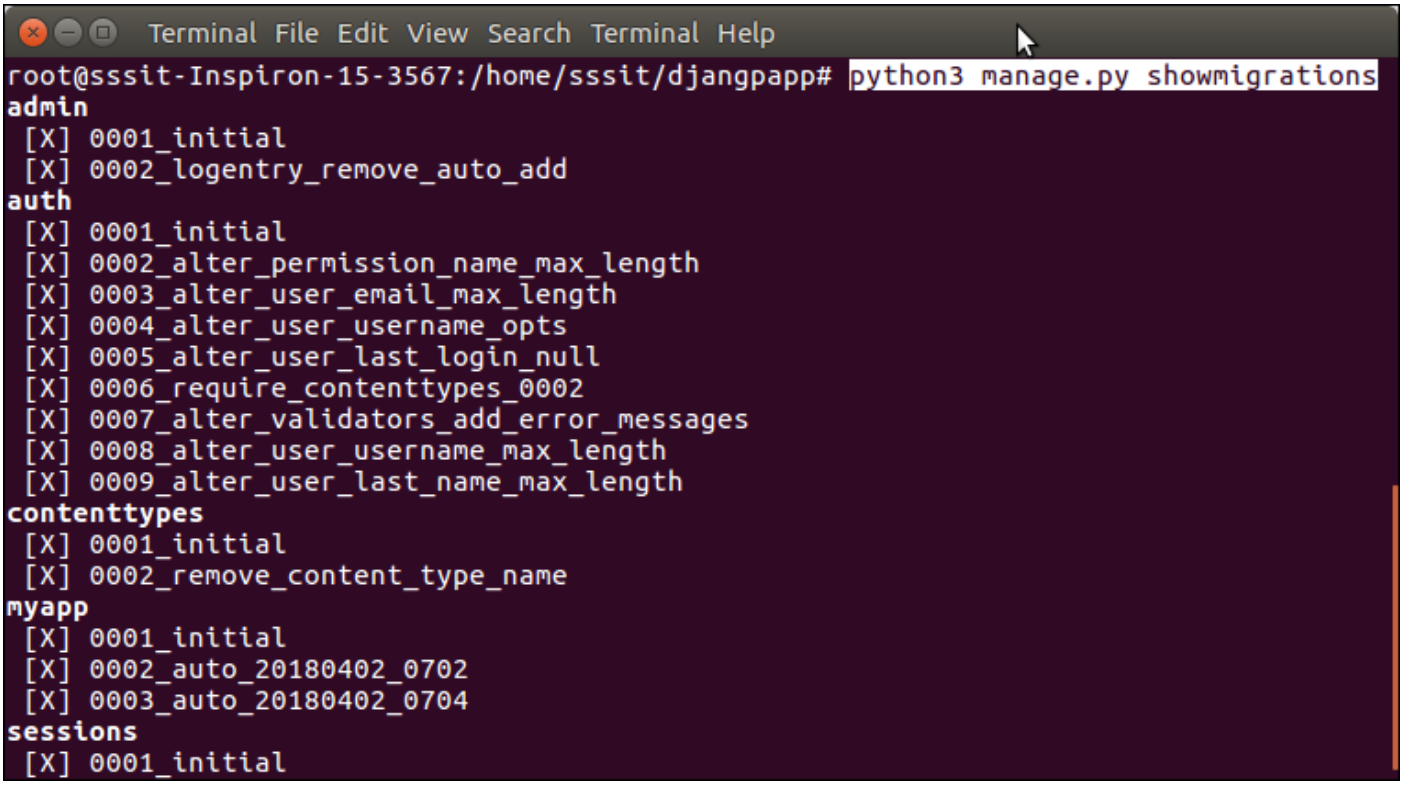
$ python3 manage.py migrate



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

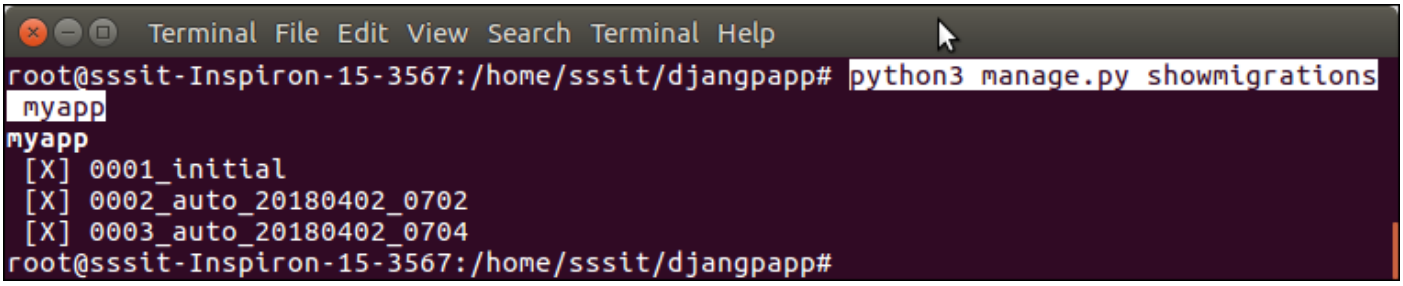
If no app-name is provided, it shows all migrations applied to the project.

$ python3 manage.py showmigrations



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

$ python3 manage.py showmigrations myapp



Django Middleware

In Django, middleware is a lightweight plugin that processes during request and response execution. Middleware is used to perform a function in the application. The functions can be a 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 is used to provides functionalities to the application. For example, Security Middleware is used to maintain the security of the application.

// settings.py

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
```

```
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Creating Own Middleware

middleware is a class that takes an argument **get_response** and returns a response.

```
class FirstMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        response = self.get_response(request)
        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.

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XframeOptionsMiddleware',
  'add new created middleware here'
]
```

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 passes 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 to our middleware.

process_view(request, view_func, view_args, view_kwargs)

It takes HttpRequest object, function object, list of arguments passed to the view or a 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 a 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 a response, the middleware will be applied and the result returns to the browser. Otherwise, the exception is handle by default handling system.

Django Request and Response

The client-server architecture includes two major components request and response. The Django framework uses client-server architecture to implement web applications.

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.

Let's have a look at the HttpRequest class.

Django HttpRequest

This class is defined in the **django.http** module and used to handle the 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 the 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

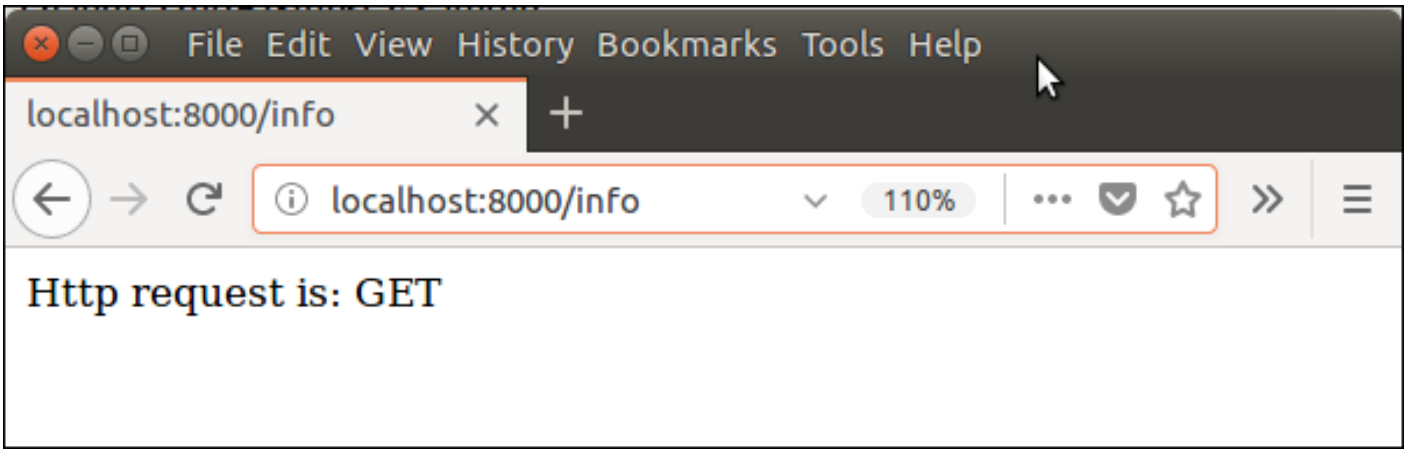
```
def methodinfo(request):
    return HttpResponse("Http request is: "+request.method)
```

// urls.py

```
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 corresponds 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 the response in the Django application.

Django Exceptions

An exception is an 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 raises 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 the database is not available.
IntegrityError	It occurs when an insertion query executes.
DataError	It raises when data related issues come into the database.

Django Http Exceptions

The following exceptions are 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 a DoesNotExist exception if data not found. This is Django's built-in exception.

// views.py

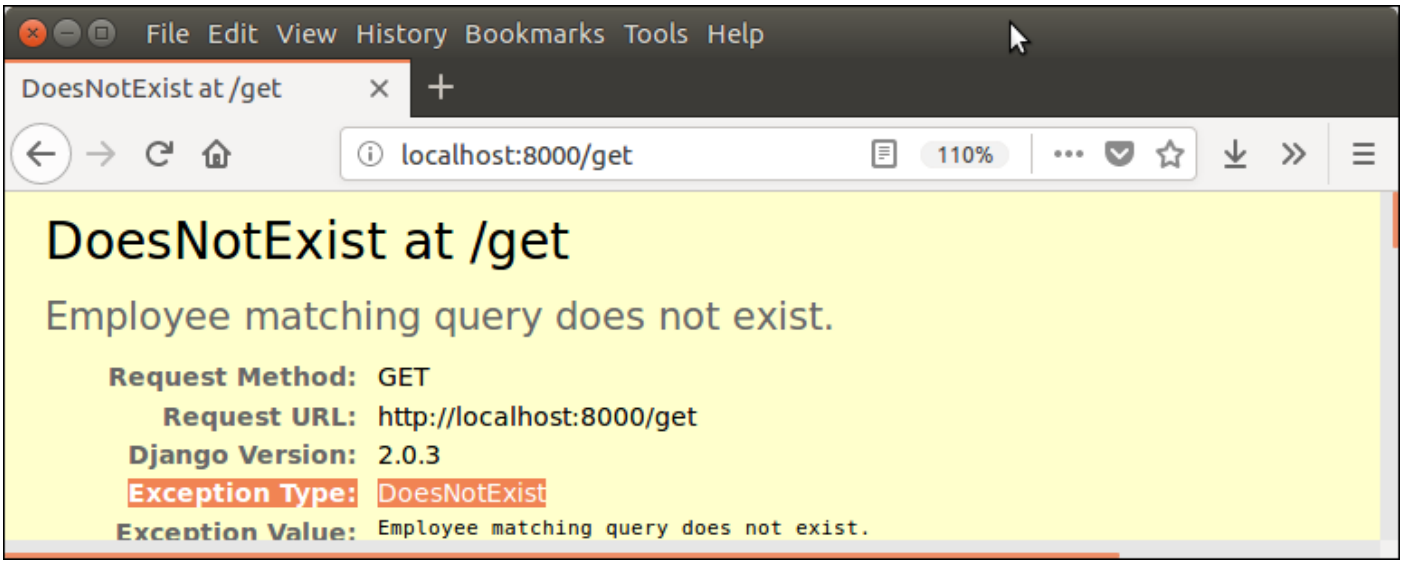
```
def getdata(request):
    data = Employee.objects.get(id=12)
    return HttpResponse(data)
```

// urls.py

path('get',views.getdata)

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

Output:

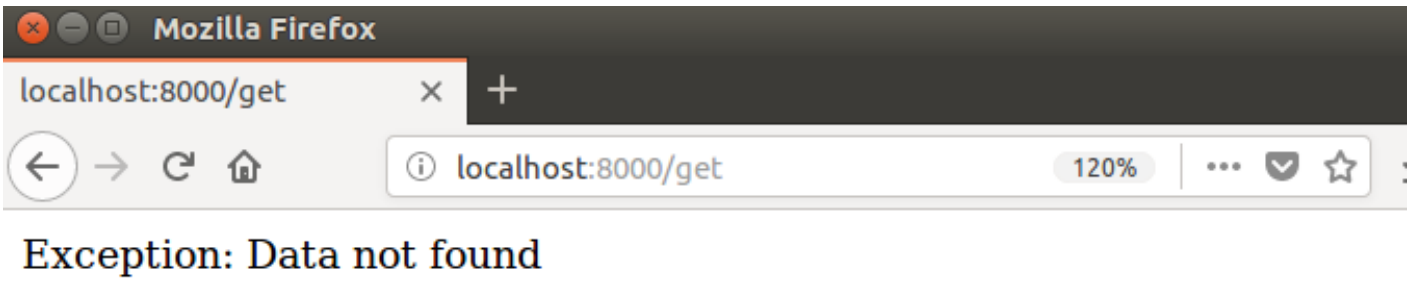


We can handle it by using try and except, now let's handle this exception.

// Views.py

```
def getdata(request):
    try:
        data = Employee.objects.get(id=12)
    except ObjectDoesNotExist:
        return HttpResponse("Exception: Data not found")
    return HttpResponse(data);
```

Output:



Django Session

A session is a mechanism to store information on the server side during the interaction with the web application.

In Django, by default session stores in the 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 the 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 the 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 checks whether the container contains the particular session object or not.
get(key, default=None)	It is used to get session value of the specified key.

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

Django Session Example

The first function is used to set and the second is used to get session values.

//views.py

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

def setsession(request):
    request.session['sname'] = 'irfan'
    request.session['semail'] = 'irfan.sssit@gmail.com'
    return HttpResponse("session is set")
def getsession(request):
    studentname = request.session['sname']
    studentemail = request.session['semail']
    return HttpResponse(studentname+" "+studentemail);
```

Url mapping to call both the functions.

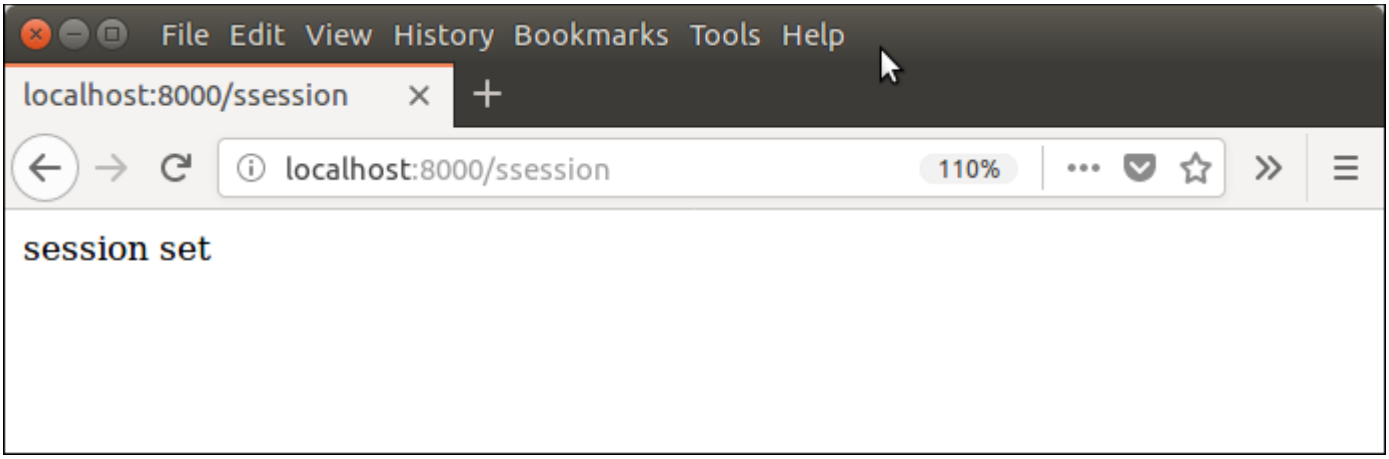
// urls.py

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
    path('ssession',views.setsession),
    path('gsession',views.getsession)
]
```

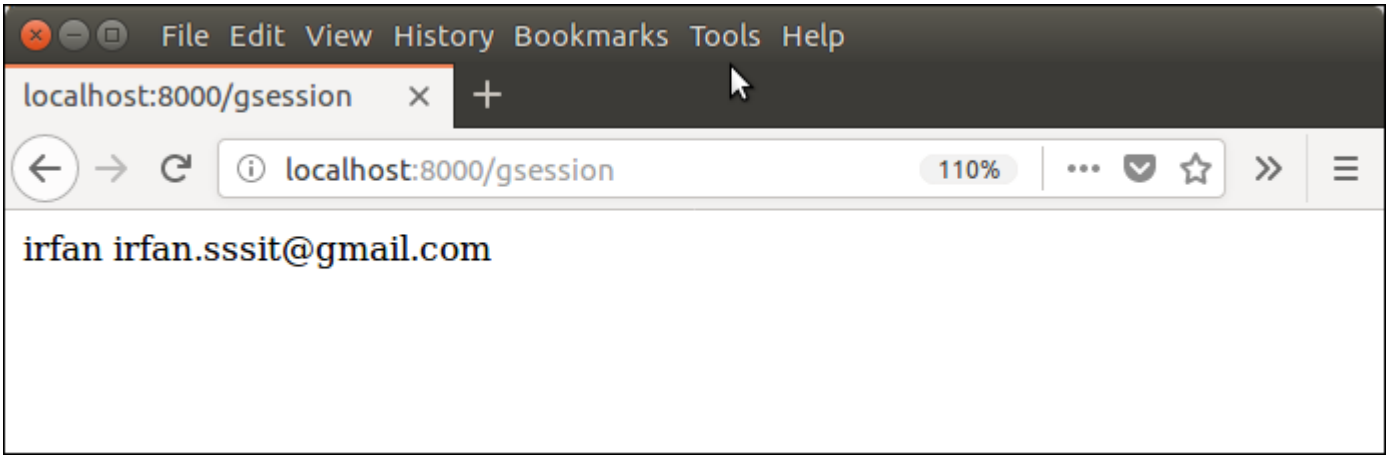
Run Server

$ python3 manage.py runserver

And set the session by using **localhost:8000/ssession**



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



Django Cookie

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

Cookie has its expiry date and time and removes automatically when gets expire. Django provides built-in methods to set and fetch cookie.

The **set_cookie()** method is used to set a cookie and **get()** method is used to get the 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

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

def setcookie(request):
    response = HttpResponse("Cookie Set")
    response.set_cookie('java-tutorial', 'javatpoint.com')
    return response
def getcookie(request):
    tutorial  = request.COOKIES['java-tutorial']
    return HttpResponse("java tutorials @: "+  tutorial);
```

And URLs specified to access these functions.

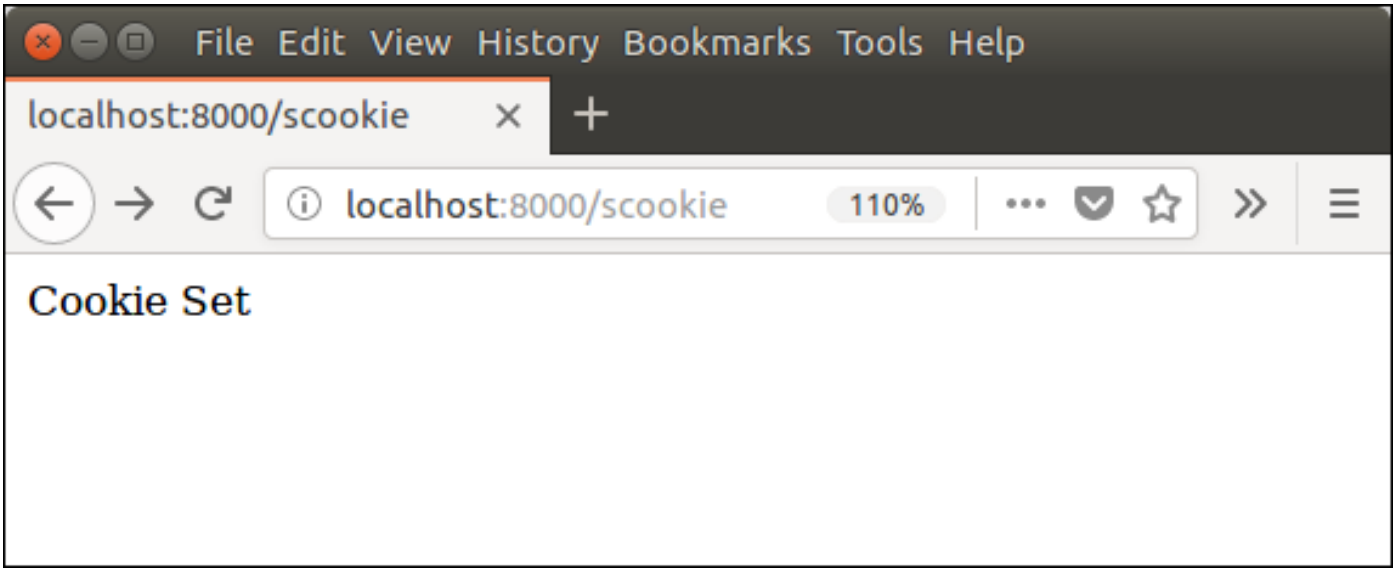
// urls.py

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('index/', views.index),
    path('scookie',views.setcookie),
    path('gcookie',views.getcookie)
]
```

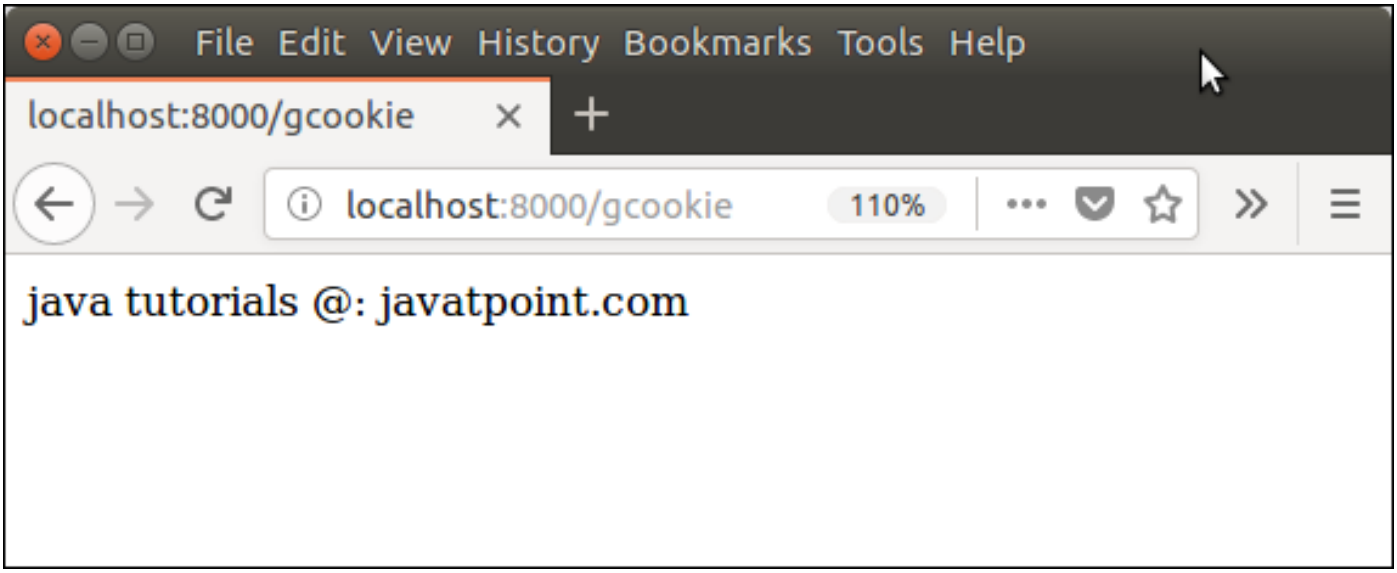
Start Server

$ python3 manage.py runserver

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



And get a 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 in our project's view file.

Let's see an example, here we have a Django project to which 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

```
import csv

def getfile(request):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="file.csv"'
```

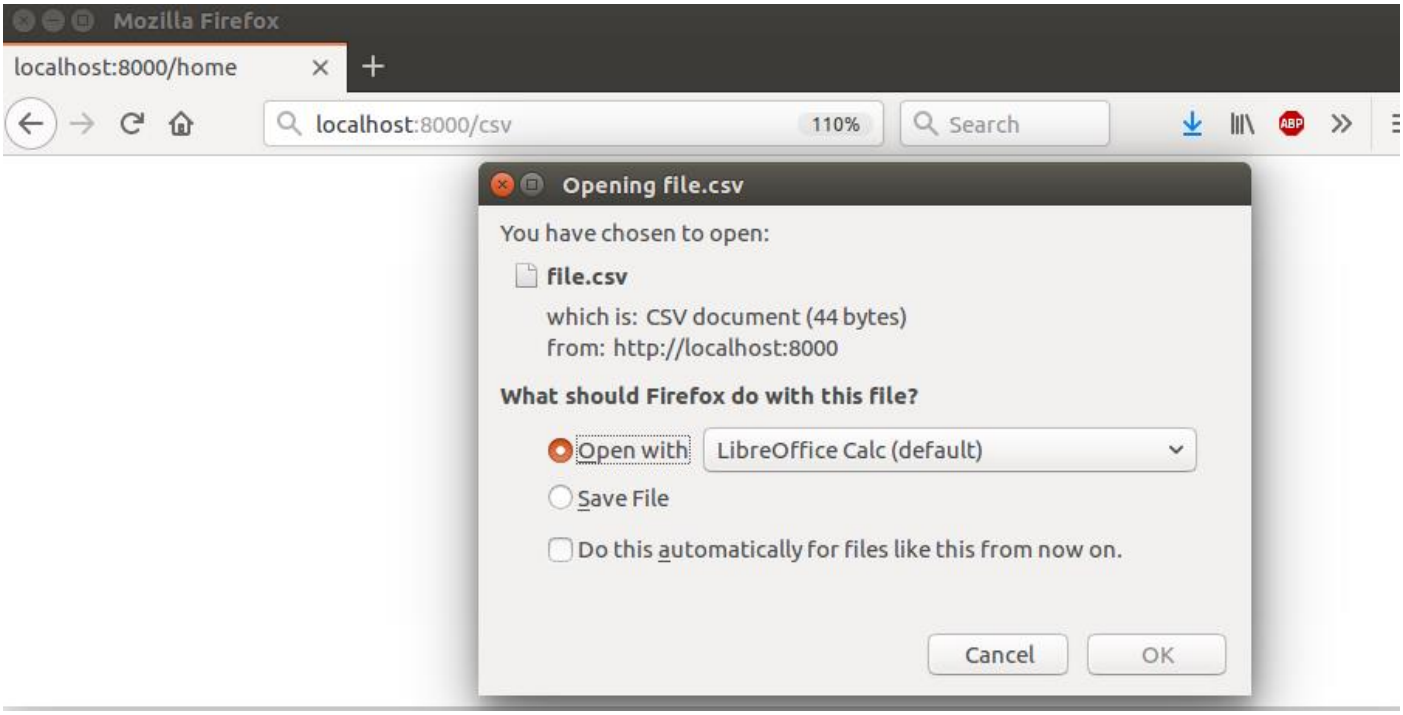
```
    writer = csv.writer(response)
    writer.writerow(['1001', 'John', 'Domil', 'CA'])
    writer.writerow(['1002', 'Amit', 'Mukharji', 'LA', '"Testing"'])
    return response
```

// urls.py

Provide url for the function.

path('csv',views.getfile)

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



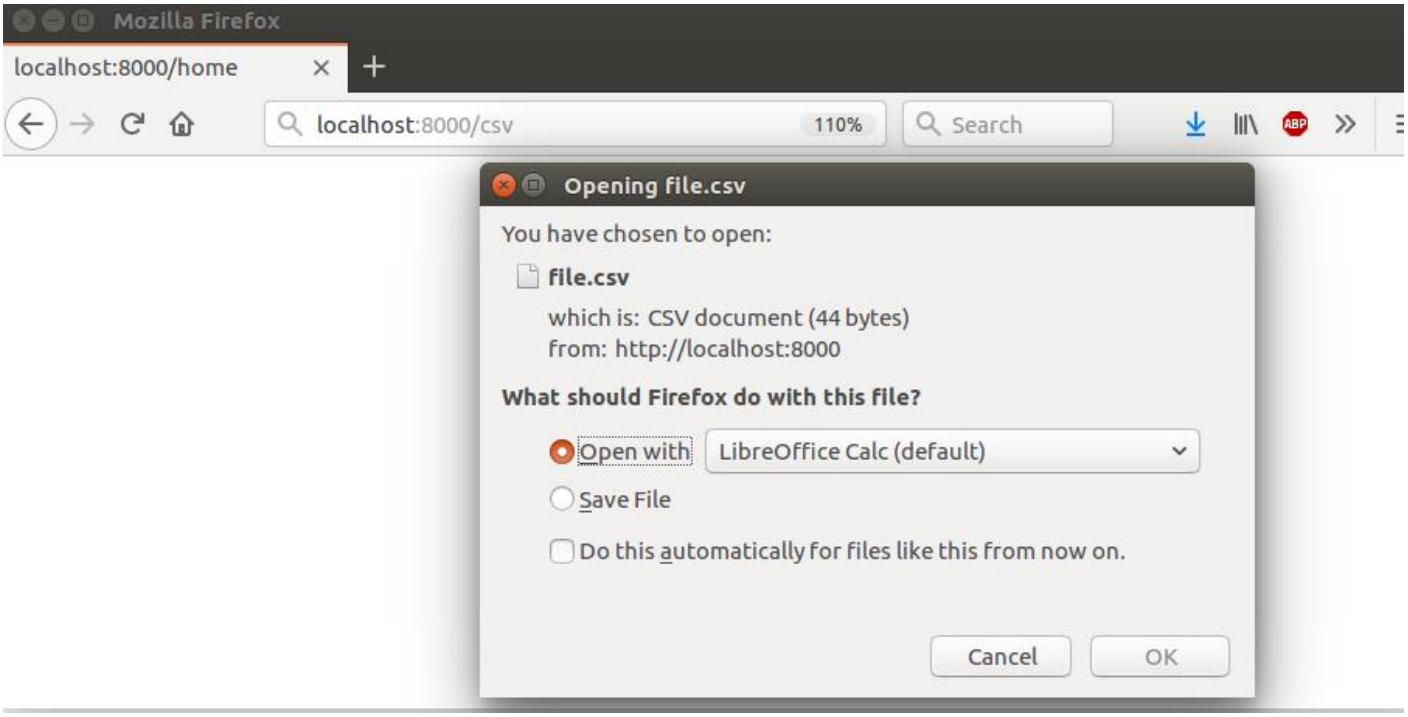
Apart from static data, we can get CSV from the database too. See, the following example in which we are getting data from the table by using the **Employee** model.

Dynamic CSV using Database

// views.py

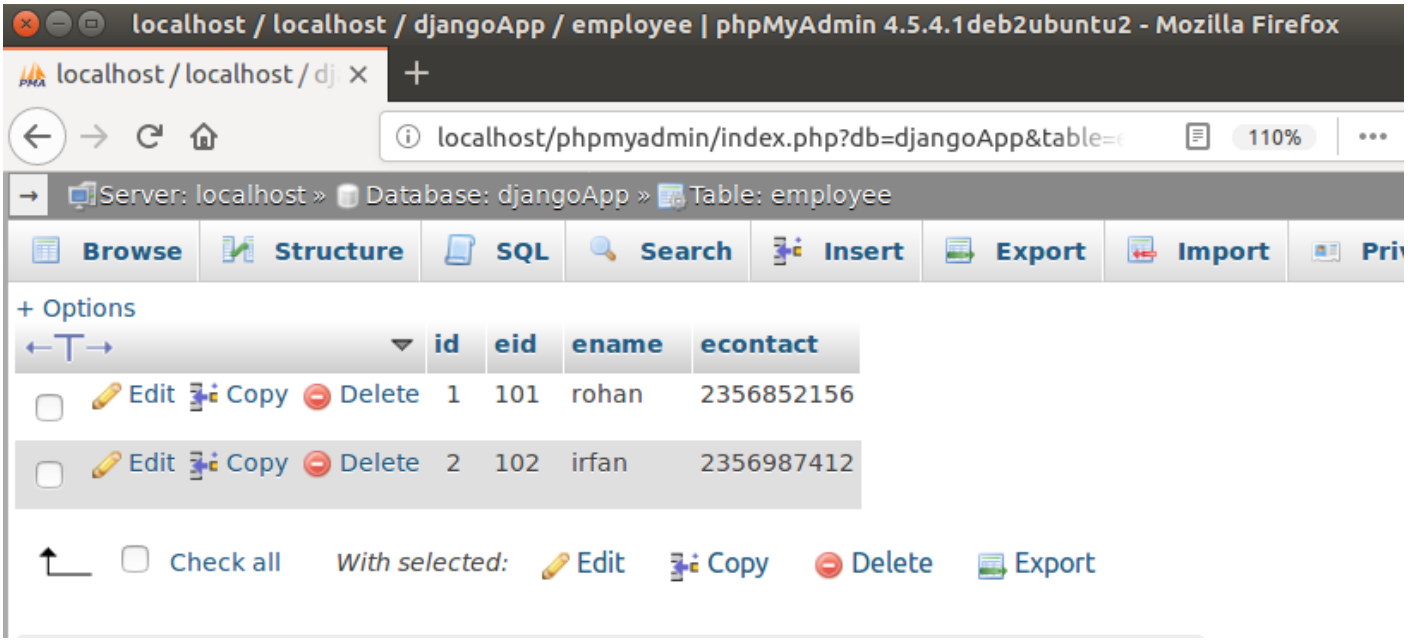
```
from myapp.models import Employee import csv
def getfile(request):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="file.csv"'
    employees = Employee.objects.all()
    writer = csv.writer(response)
    for employee in employees:
        writer.writerow([employee.eid,employee.ename,employee.econtact])
    return response
```

Output:



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

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



Well, we have seen that this library is very useful to create a 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.

$ 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 customized PDF. See the example.

// views.py

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def getpdf(request):
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="file.pdf"'
    p = canvas.Canvas(response)
    p.setFont("Times-Roman", 55)
    p.drawString(100,700, "Hello, Javatpoint.")
    p.showPage()
    p.save()
    return response
```

First, provide MIME (content) type as application/pdf, so that output generates as PDF rather than 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 to the save() method and return response.

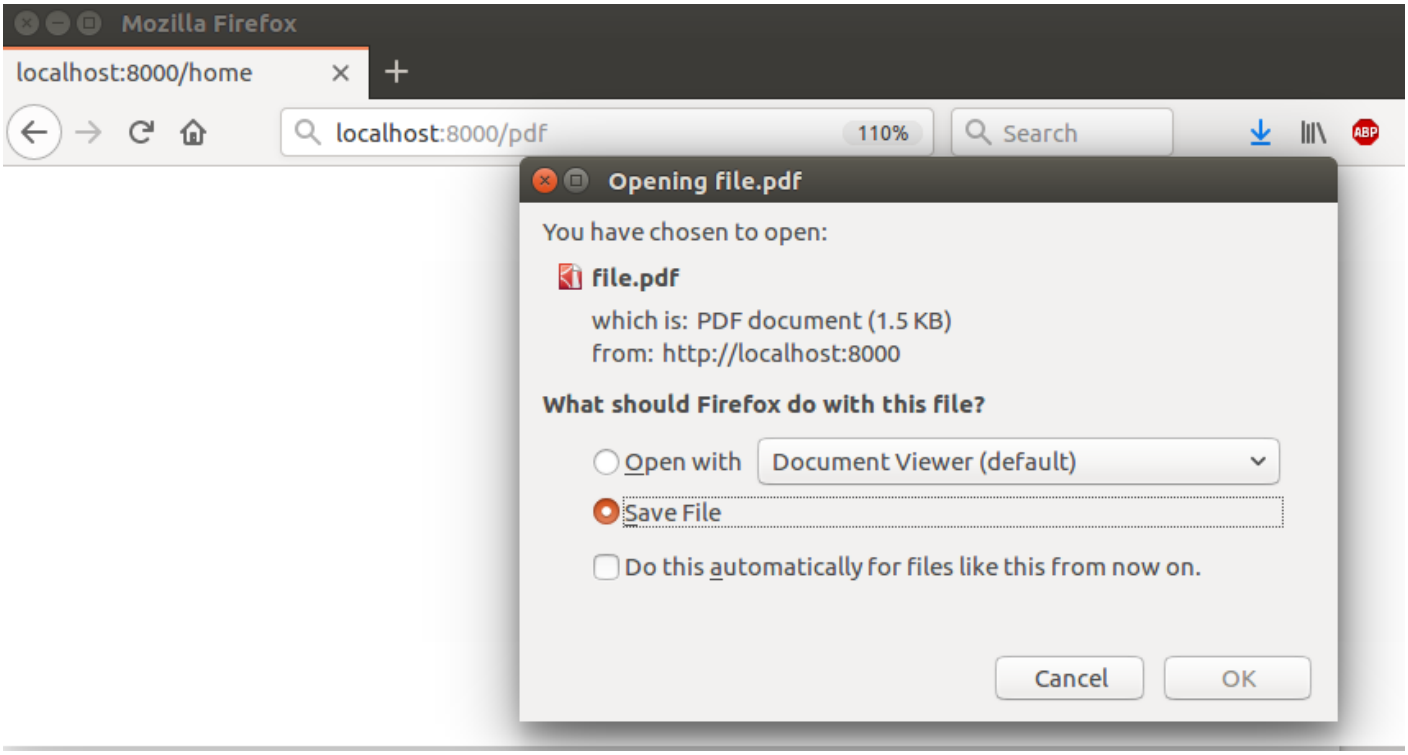
// urls.py

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

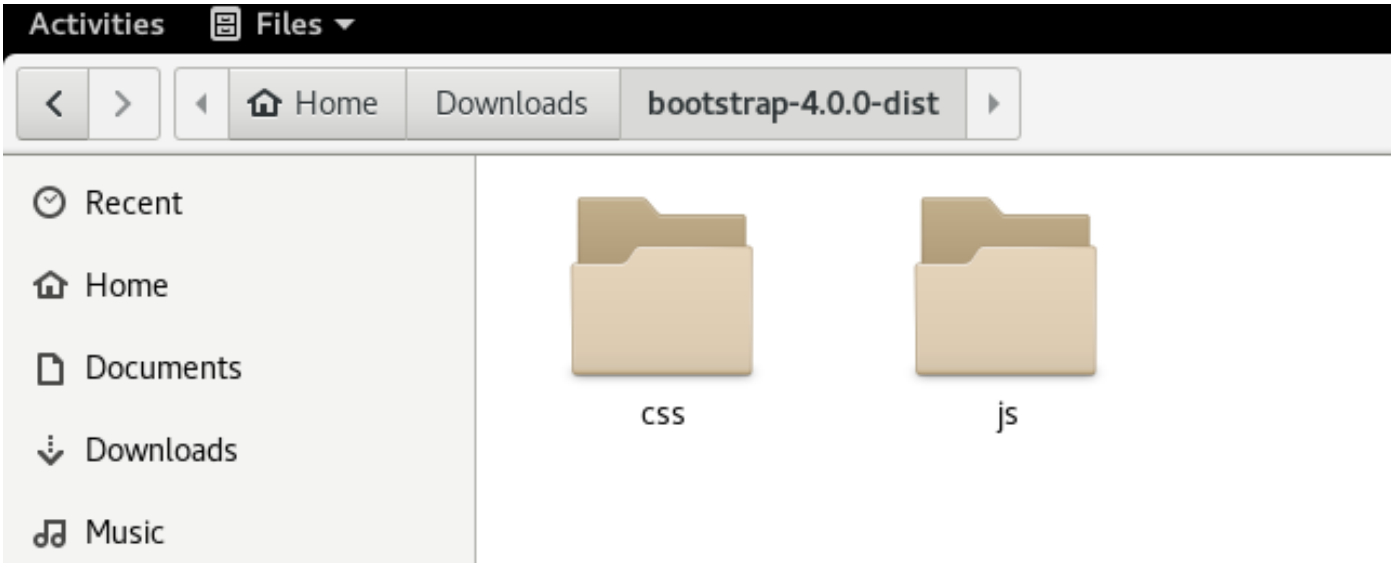
Bootstrap is a framework which is used to create user interface in web applications. It provides css, js and other tools that help to create required interface.

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

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

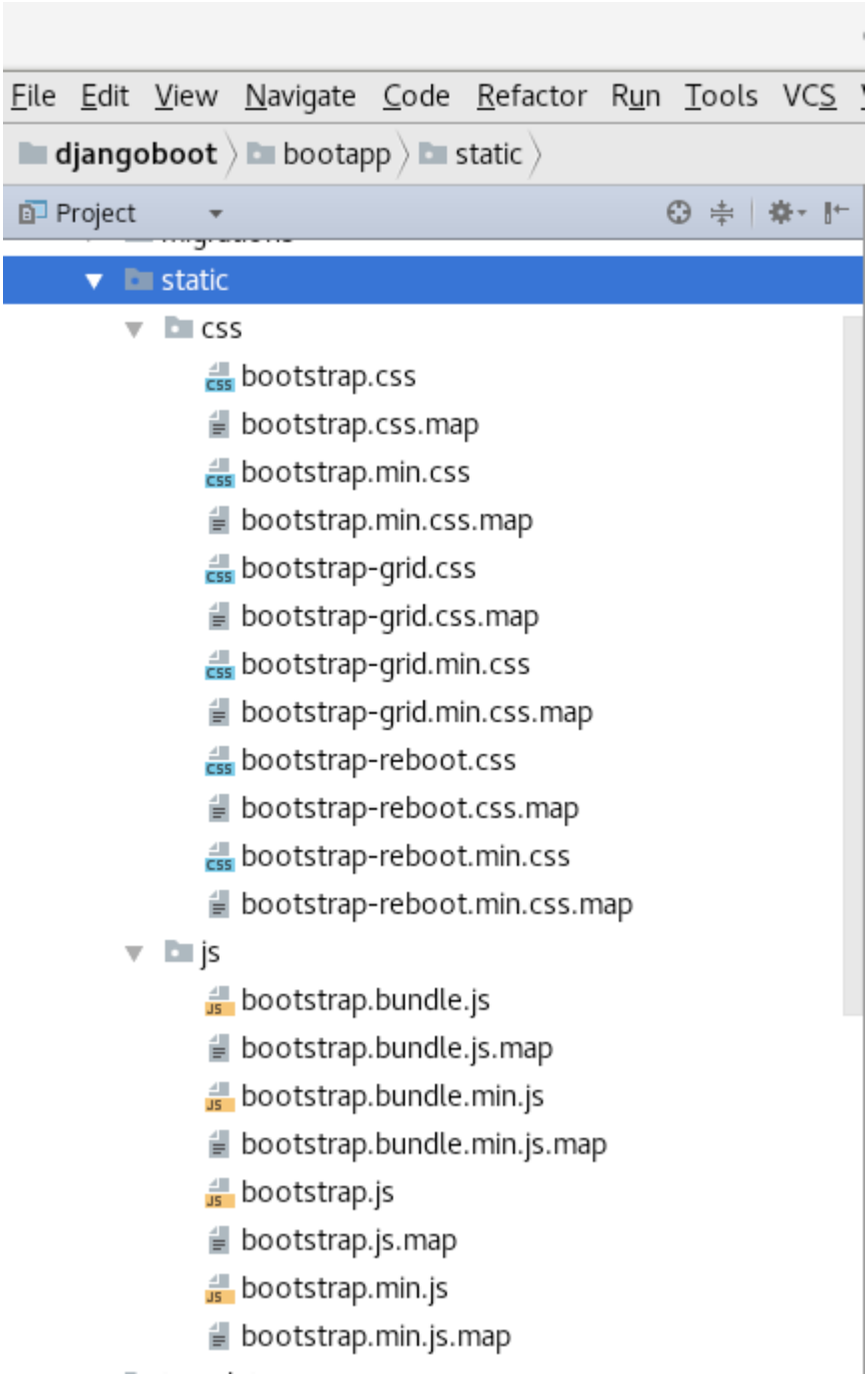
1. Download the Bootstrap

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.

{% load staticfiles %}

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

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

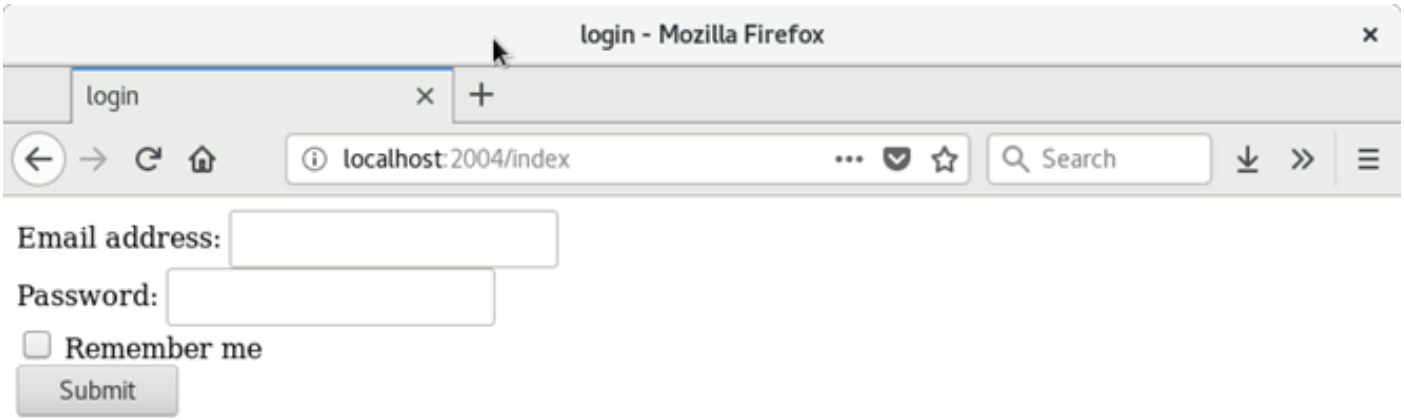
```
    <title>Title</title>
    {% load staticfiles %}
    <link href="{% static 'css/bootstrap.min.css' %}" >
    <script src="{% static 'bootstrap.min.js' %}"></script>
    <script>alert();</script>
</head>
<body>
</body>
</html>
```

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:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>login</title>
</head>
<body>
 <form action="/save" method="post">
  <div class="form-group">
    <label for="email">Email address:</label>
    <input type="email" class="form-control" id="email">
  </div>
  <div class="form-group">
    <label for="pwd">Password:</label>
    <input type="password" class="form-control" id="pwd">
  </div>
  <div class="checkbox">
    <label><input type="checkbox"> Remember me</label>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
</body>
</html>
```

Output:

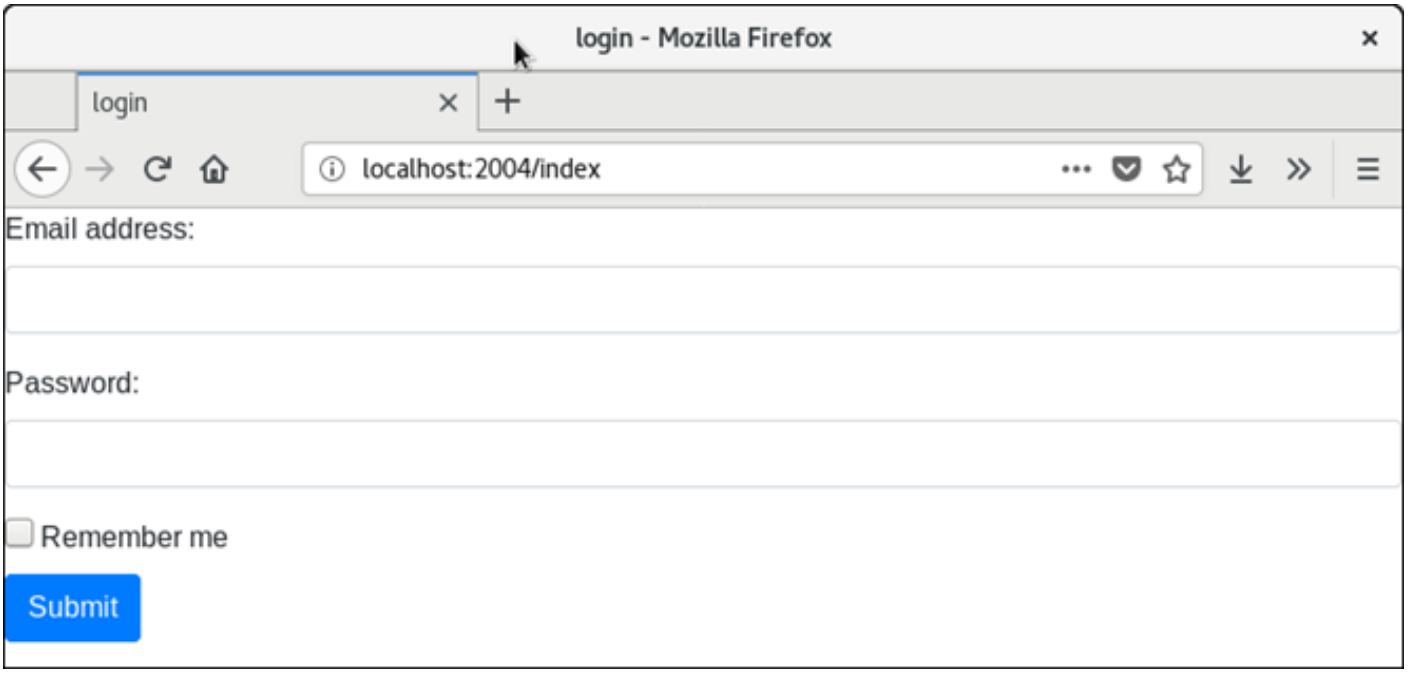


After loading bootstrap files. Our code look like this:

// index.html

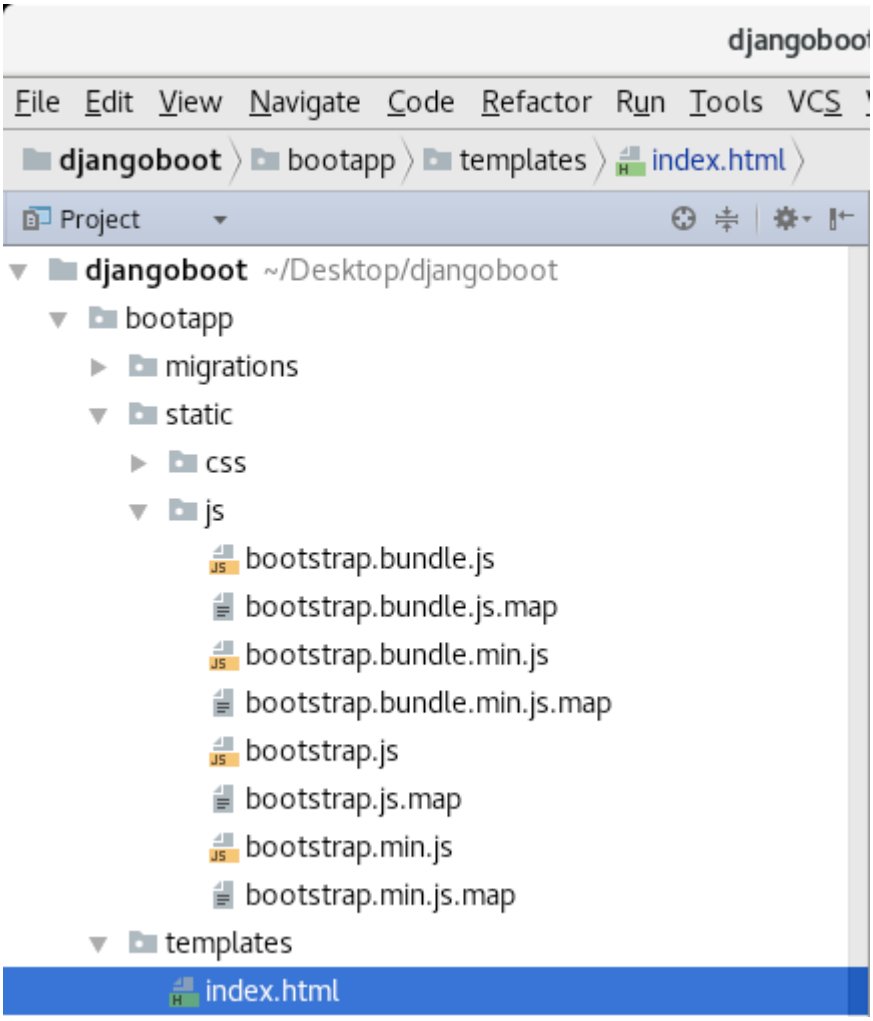
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>login</title>
    {% load staticfiles %}
    <link href="{% static 'css/bootstrap.min.css' %}" rel="stylesheet">
    <script src="{% static 'js/bootstrap.min.js' %}"></script>
</head>
<body>
 <form action="/save" method="post">
  <div class="form-group">
    <label for="email">Email address:</label>
    <input type="email" class="form-control" id="email">
  </div>
  <div class="form-group">
    <label for="pwd">Password:</label>
    <input type="password" class="form-control" id="pwd">
  </div>
  <div class="checkbox">
    <label><input type="checkbox"> Remember me</label>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
</body>
</html>
```

Output:



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

Finally, out project structure looks like this.



Django Deployment to Github

Github is a global repository system which is used for version control. While working with django, if there is need for version management, it is recommended to use github.

In this tutorial, we will create and deploy a django project to the github so that it can be accessible globally.

Before deploying, **it is required to have a github account**, otherwise create an account first by visiting github.com.

Open the terminal and **cd into the project**, we want to deploy. For example, our project name is djangoboot. Then

Install Git

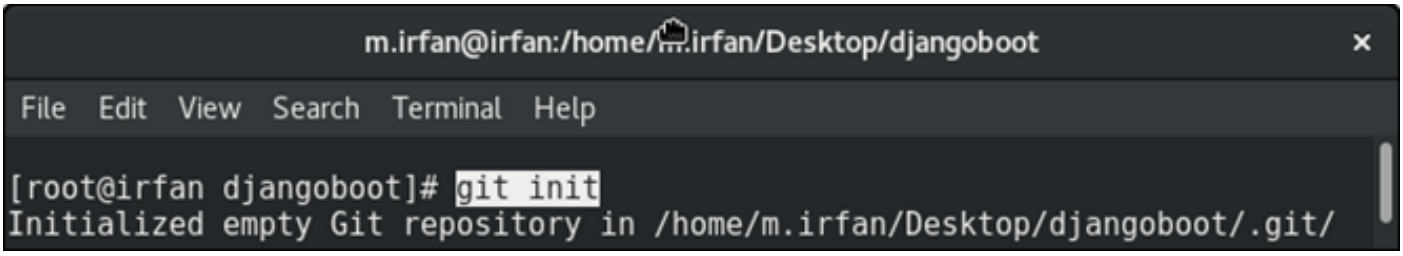
we use the following command to install git on our location machine.

$ apt-get install git

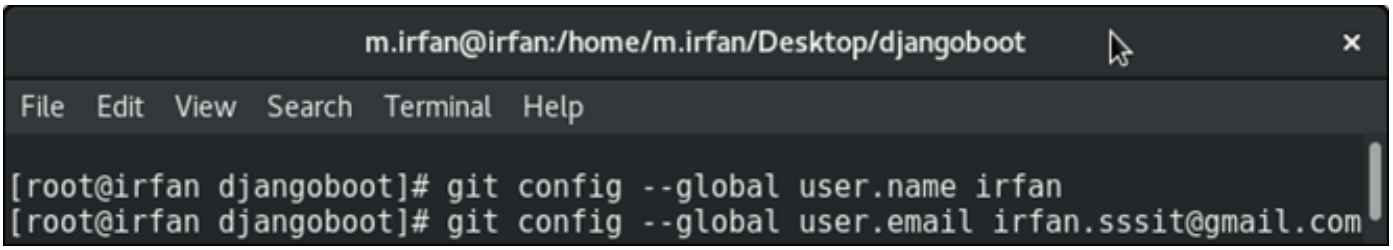
Initialize Git

Use the following command to start the git.

$ git init



Provide global user name email for the project, it is only once, we don?t need to provide it repeatedly.



Create File

Create a file **.gitignore** inside the root folder of django project. And put the following code inside it.

// .gitignore

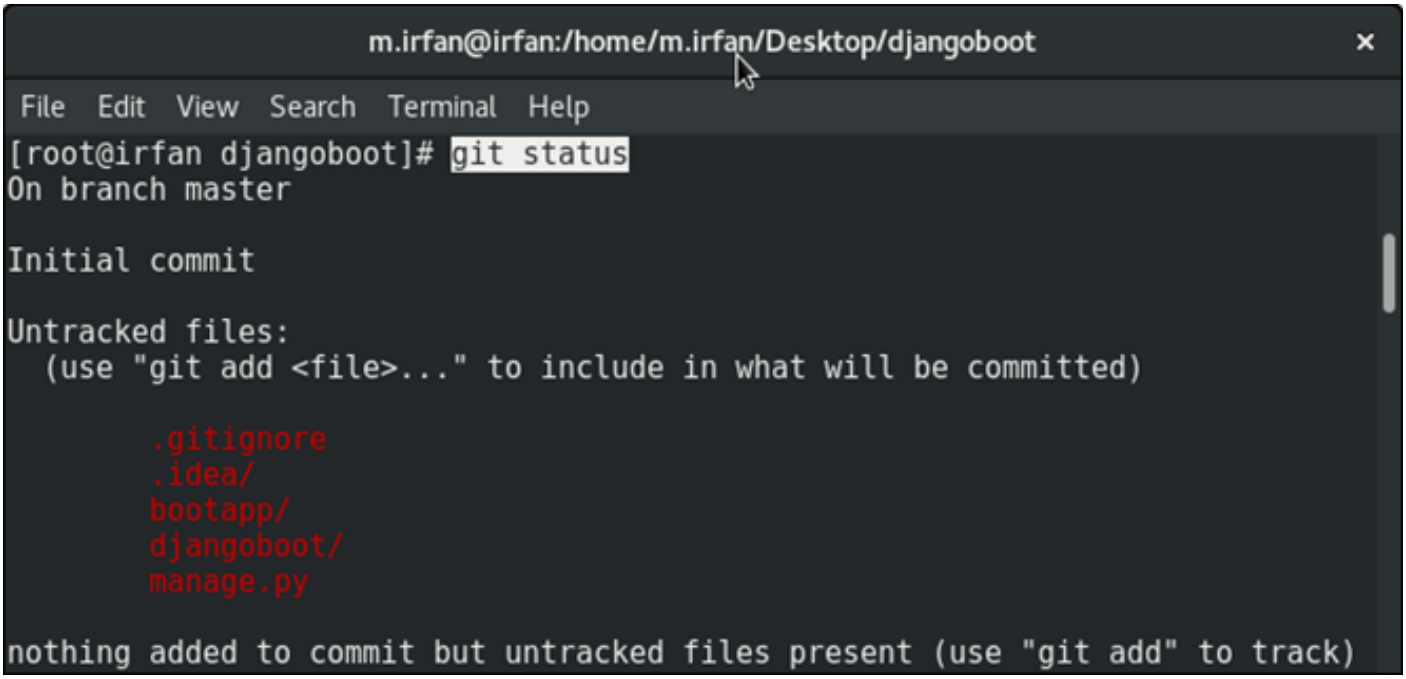
```
*.pyc
*~
__pycache__
myvenv
```

db.sqlite3
/**static**
.DS_Store

Git Status

Check the git status by using the following command. It provides some detail to the screen.

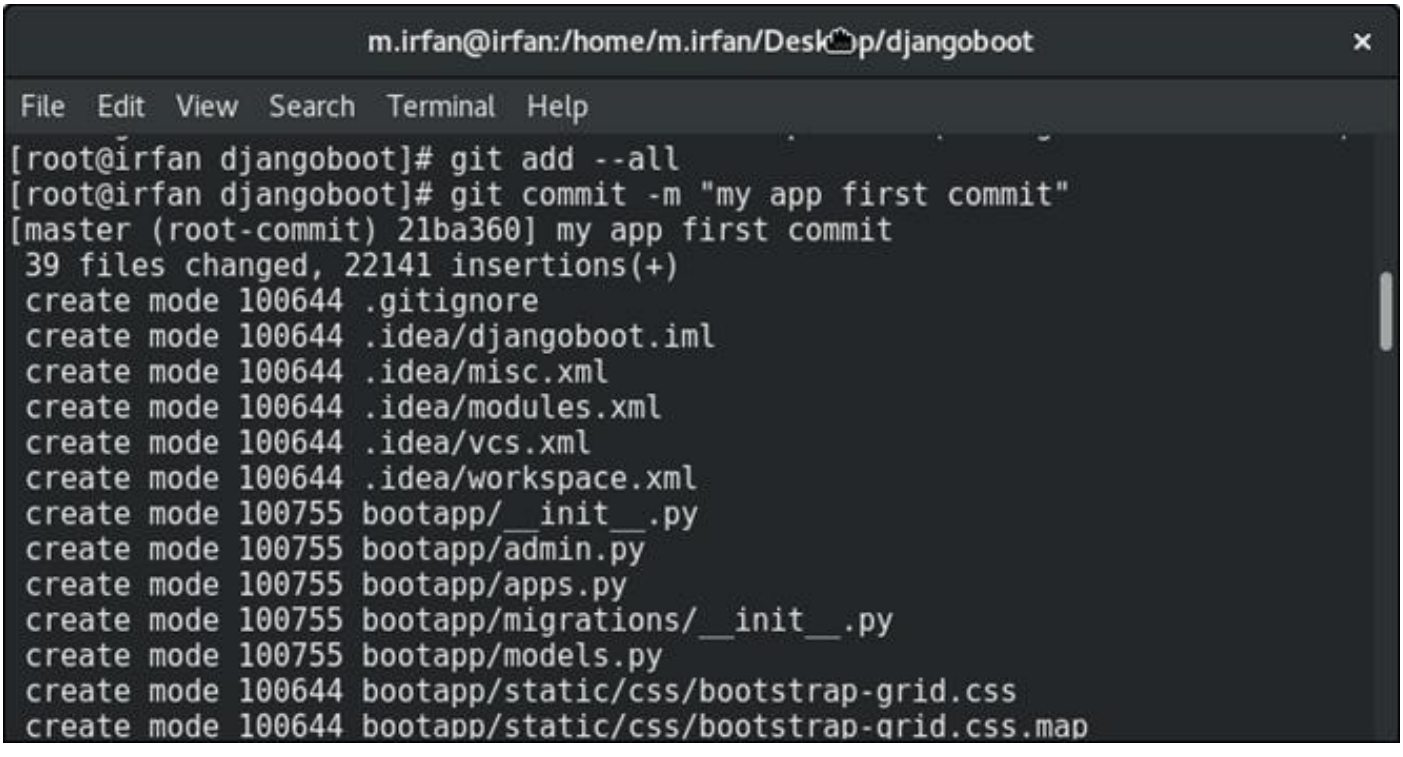
$ git status



After saving, now execute the following command.

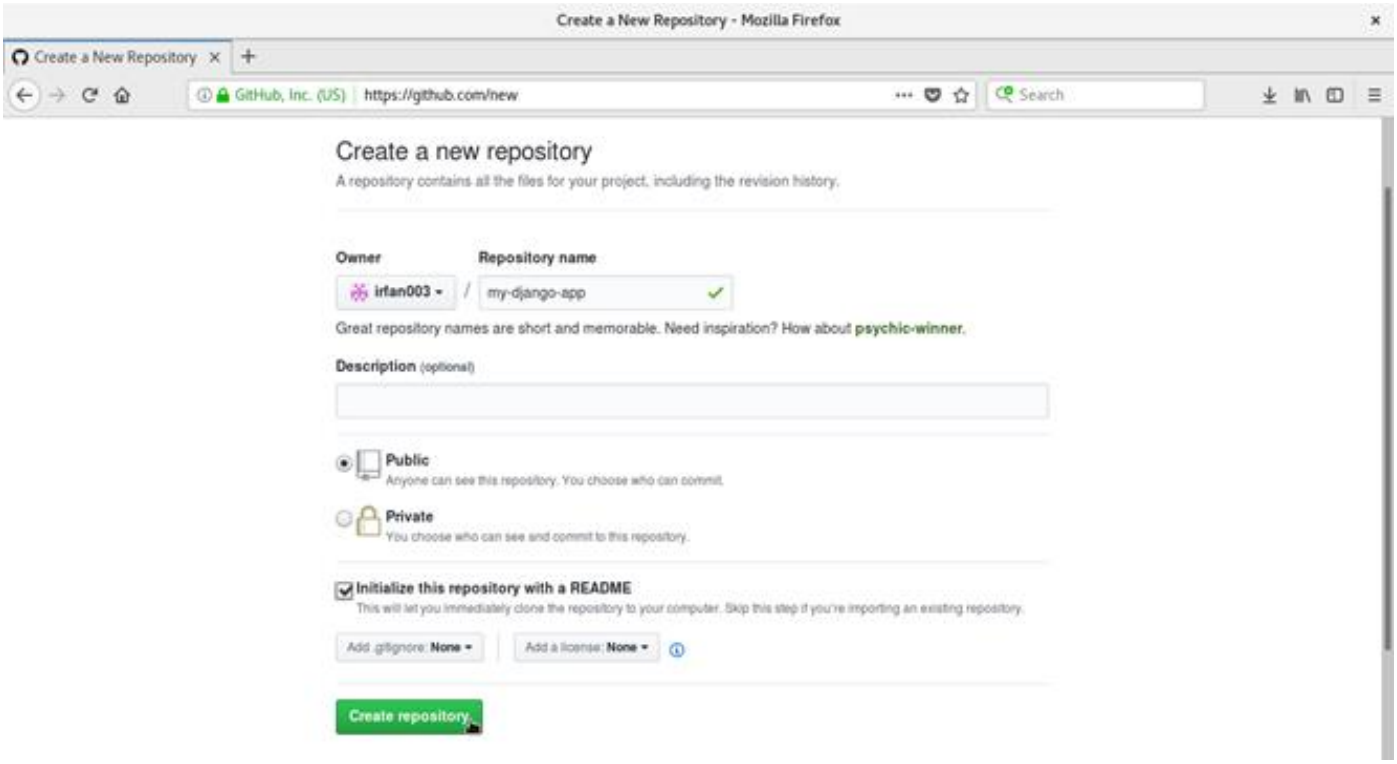
$ git add -all

$ git commit -m "my app first commit"



Push to Github

First login into the git account and create a new repository and initialize with README. See the example.



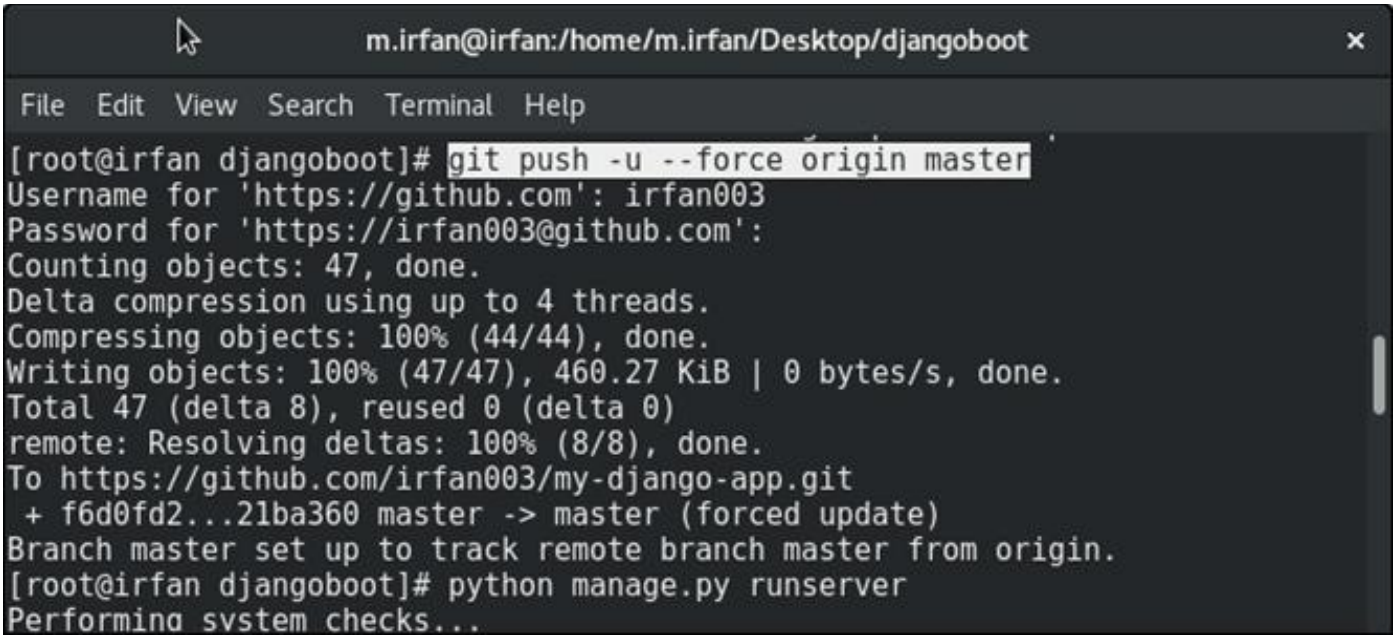
My repository name is my-django-app. Click on the create repository button. Now repository has created.

On next page, click on the clone button and copy the http url. In my case, it is **https://github.com/irfan003/my-django-app.git**

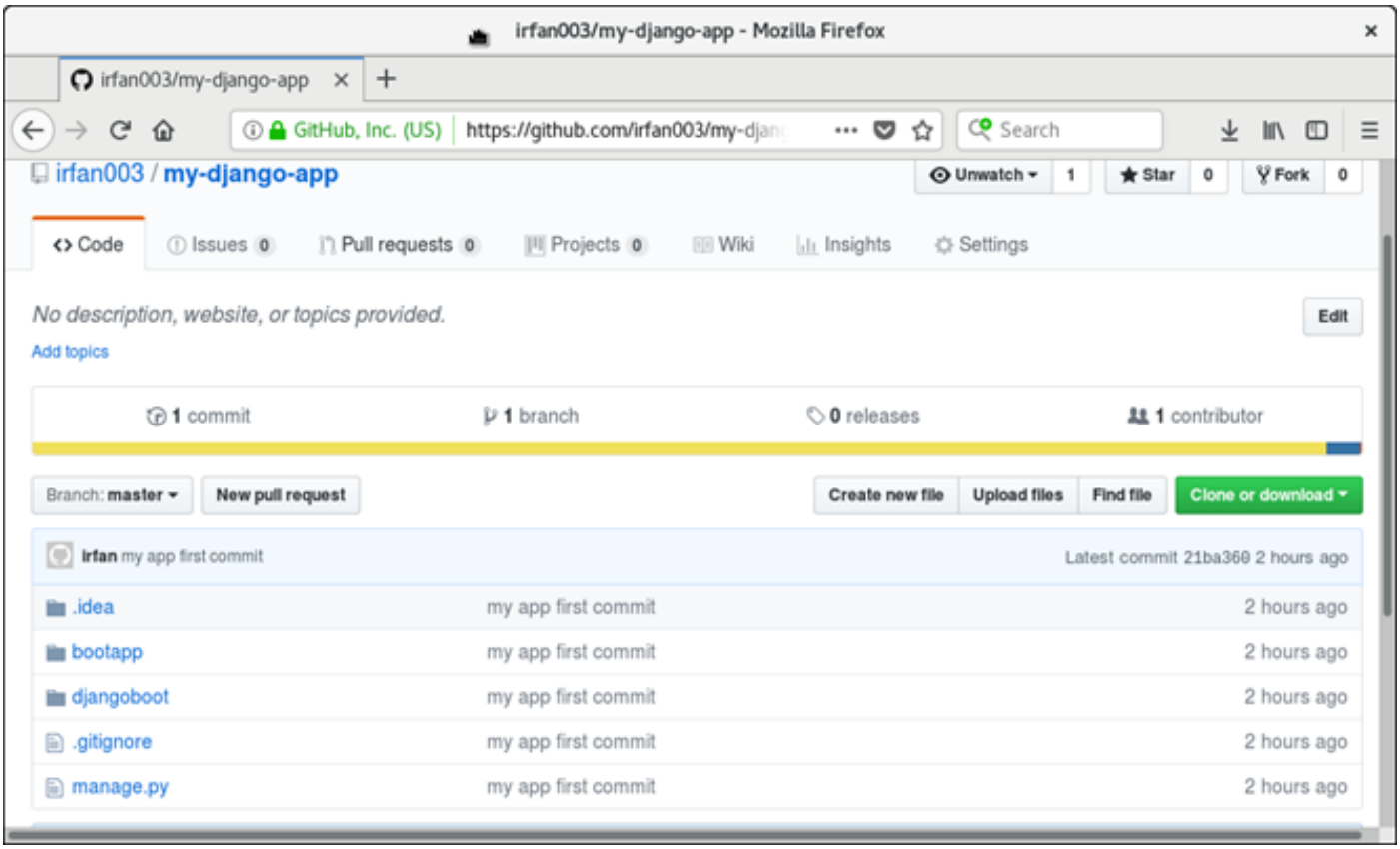
Now, use this url with the following command.

$ git remote add origin https://github.com/irfan003/my-django-app.git

$ git push -u --force origin master



provide username and password of git account. It will start pushing project to the repository. We can verify it. See the below screenshot.



See, our django application has deploy successfully on github. Now, we can access it globally.

Django Mail Setup

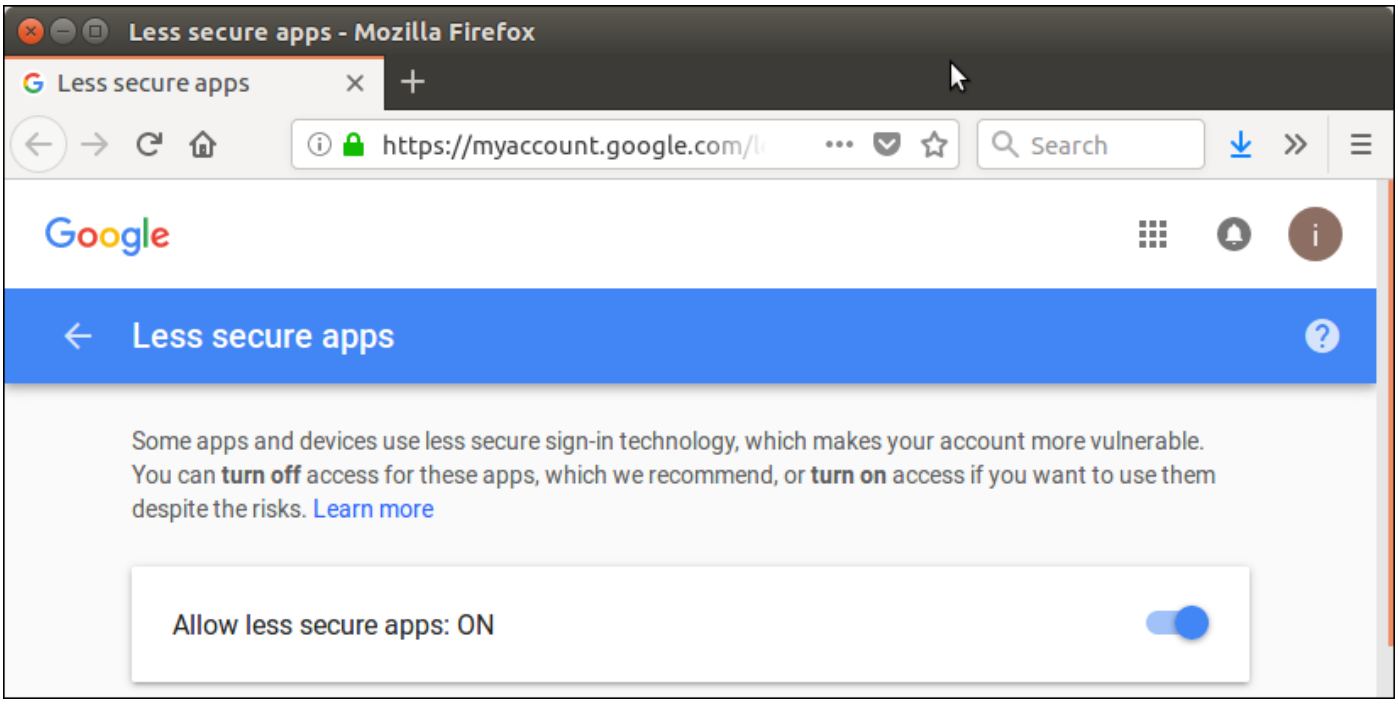
Sending email using Django is pretty easy and require less configuration. In this tutorial, we will send email to provided email.

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.

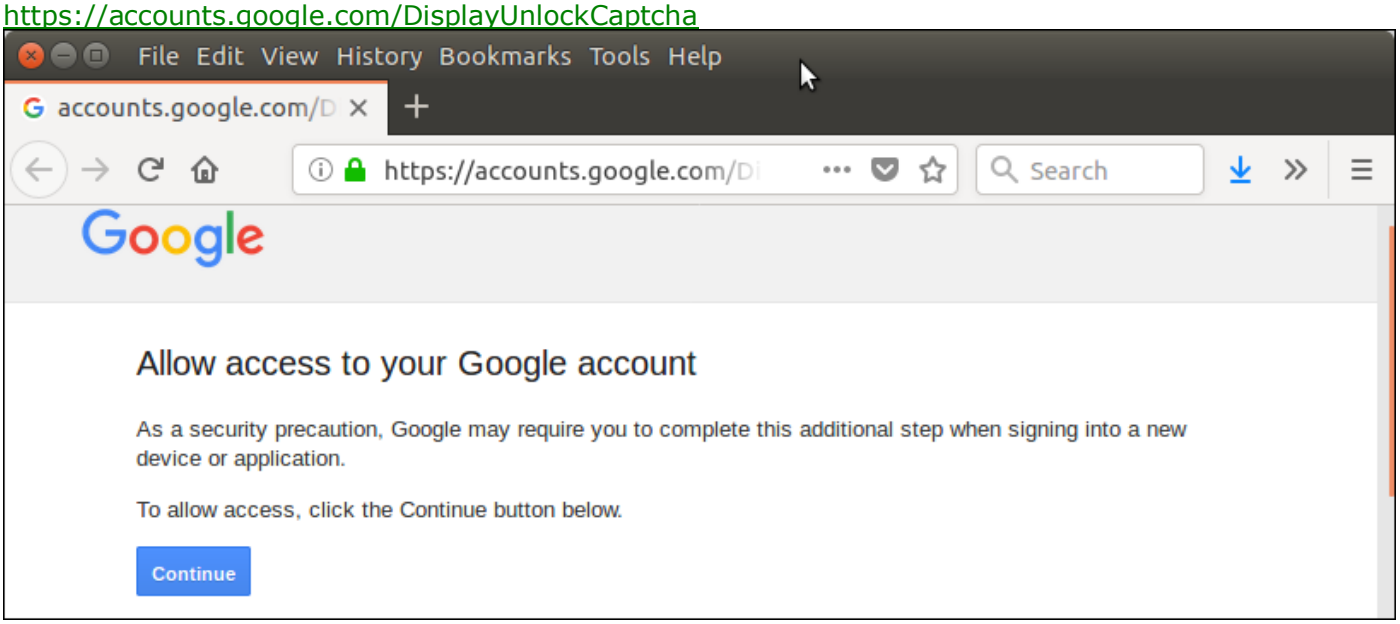
Before sending email, we need to make some changes in Gmail account because for security reasons Google does not allow direct access (login) by any application. So, login to the Gmail account and follow the urls. It will redirect to the Gmail account settings where we need to allow less secure apps but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

```
EMAIL_USE_TLS = True
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_HOST_USER = 'irfan.iit003@gmail.com'
EMAIL_HOST_PASSWORD = '*********'
```

Import Mail Library

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

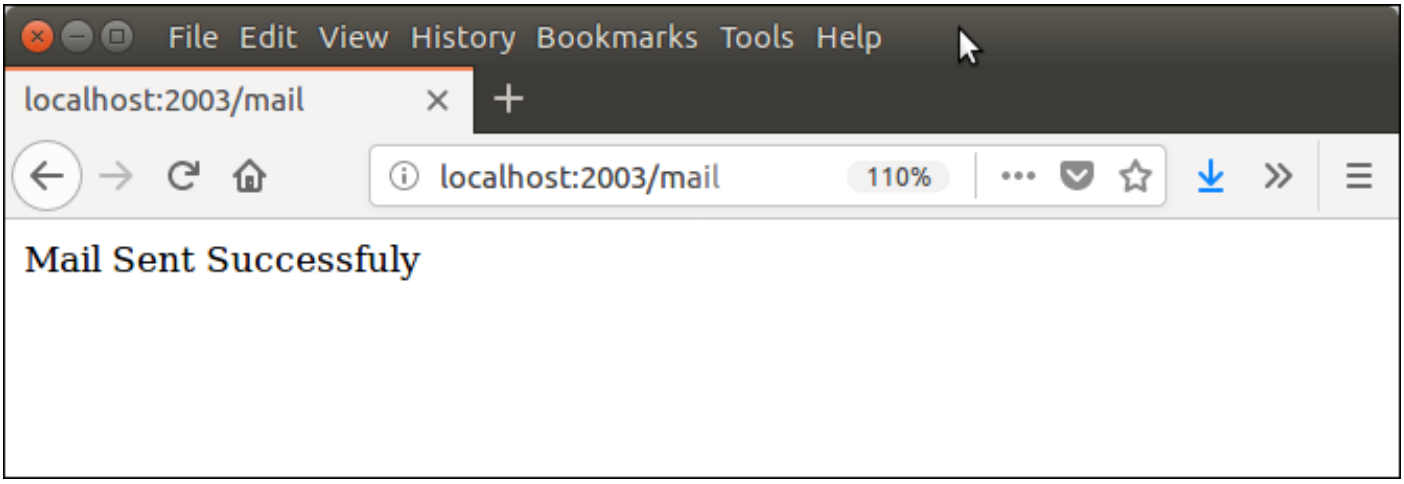
```
from django.http import HttpResponse
from djangpapp import settings
from django.core.mail import send_mail


def mail(request):
    subject = "Greetings"
    msg     = "Congratulations for your success"
    to      = "irfan.sssit@gmail.com"
    res     = send_mail(subject, msg, settings.EMAIL_HOST_USER, [to])
    if(res == 1):
        msg = "Mail Sent Successfuly"
    else:
        msg = "Mail could not sent"
    return HttpResponse(msg)
```

// urls.py

Put following url into urls.py file.

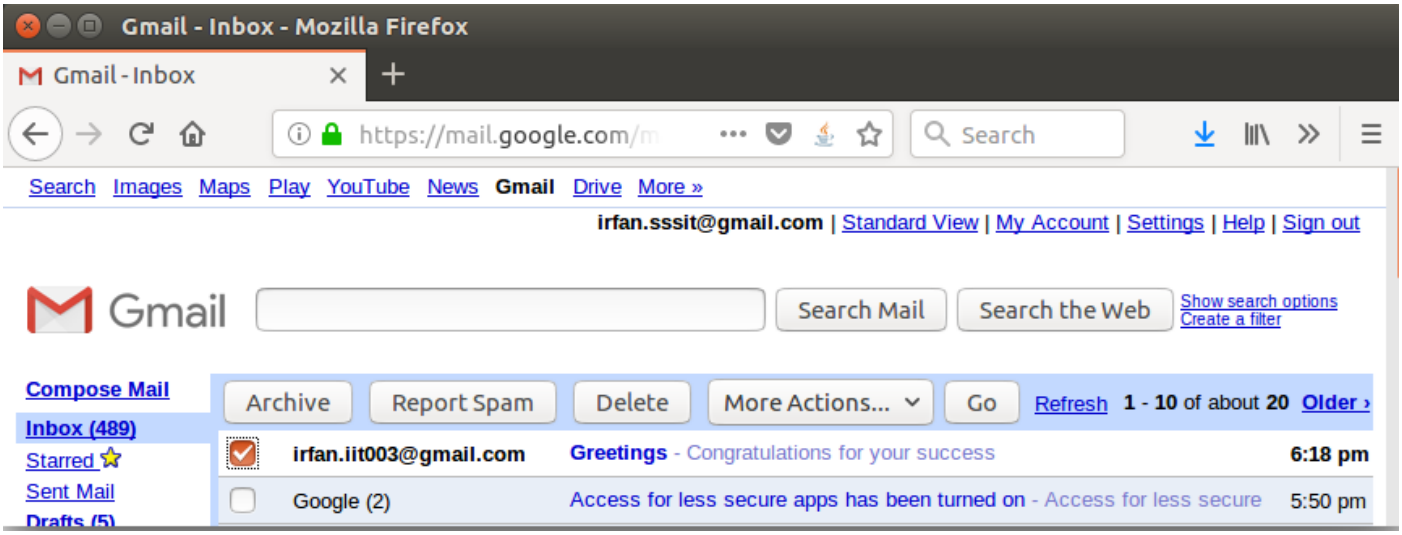
```
path('mail',views.mail)
```

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.

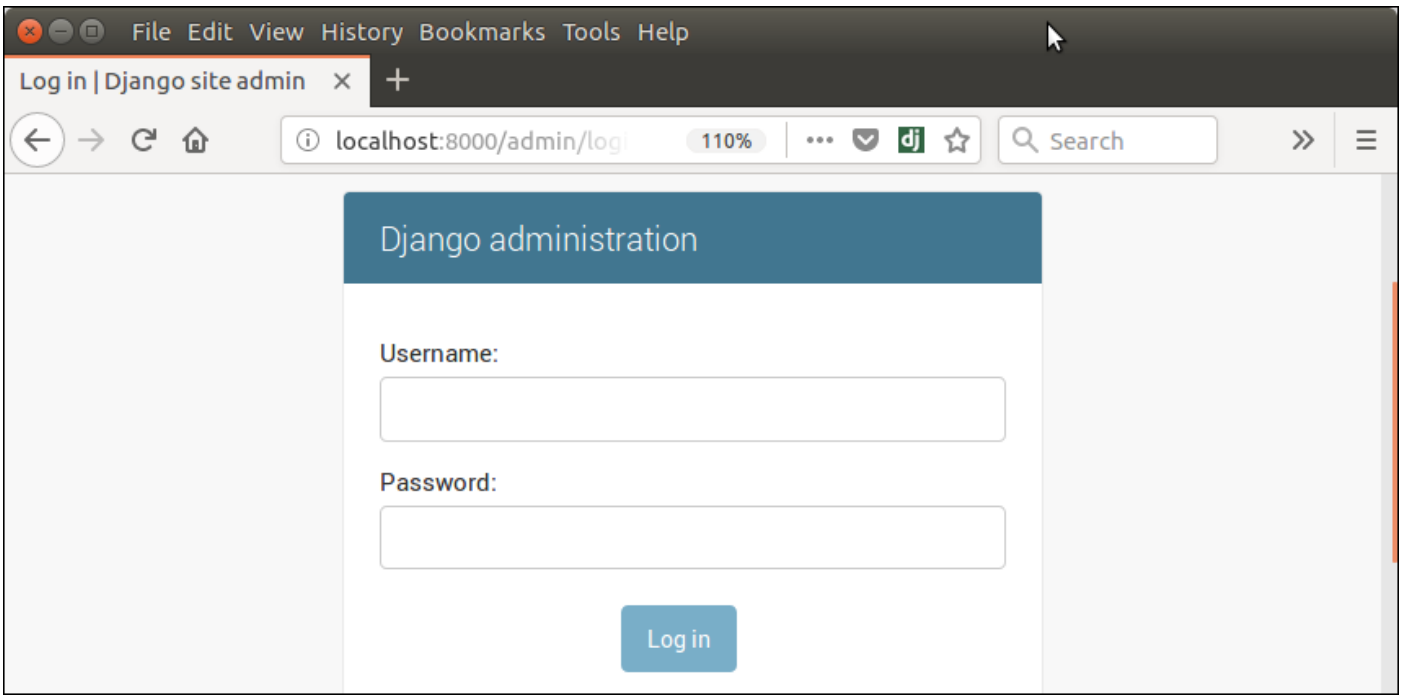
It is a built-in feature of Django that automatically generates interface for models.

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

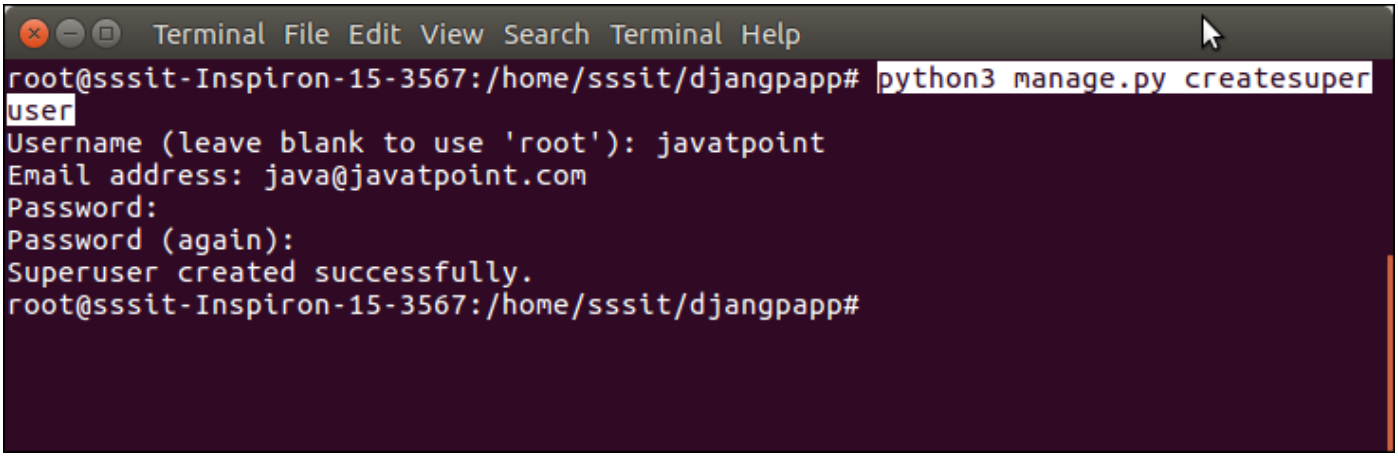
```
urlpatterns = [
    path('admin/', admin.site.urls),
]
```

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.**

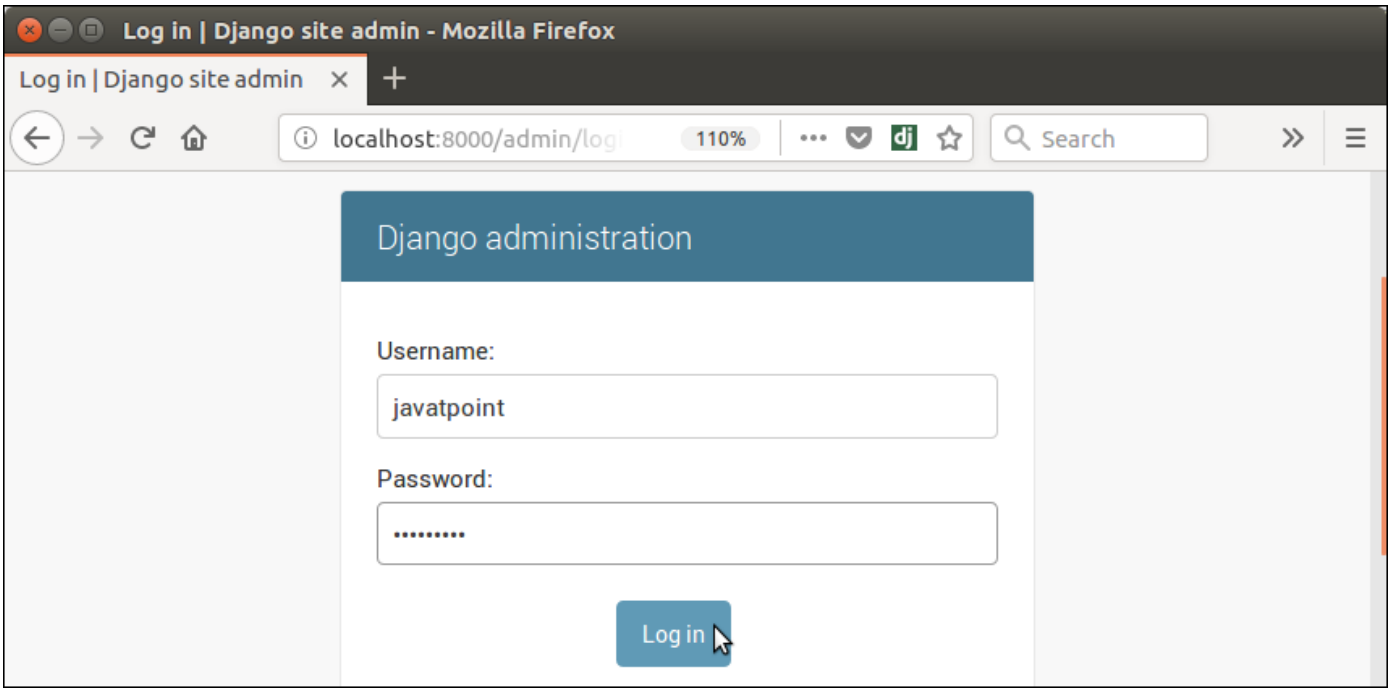
A login form will be displayed, see the below.



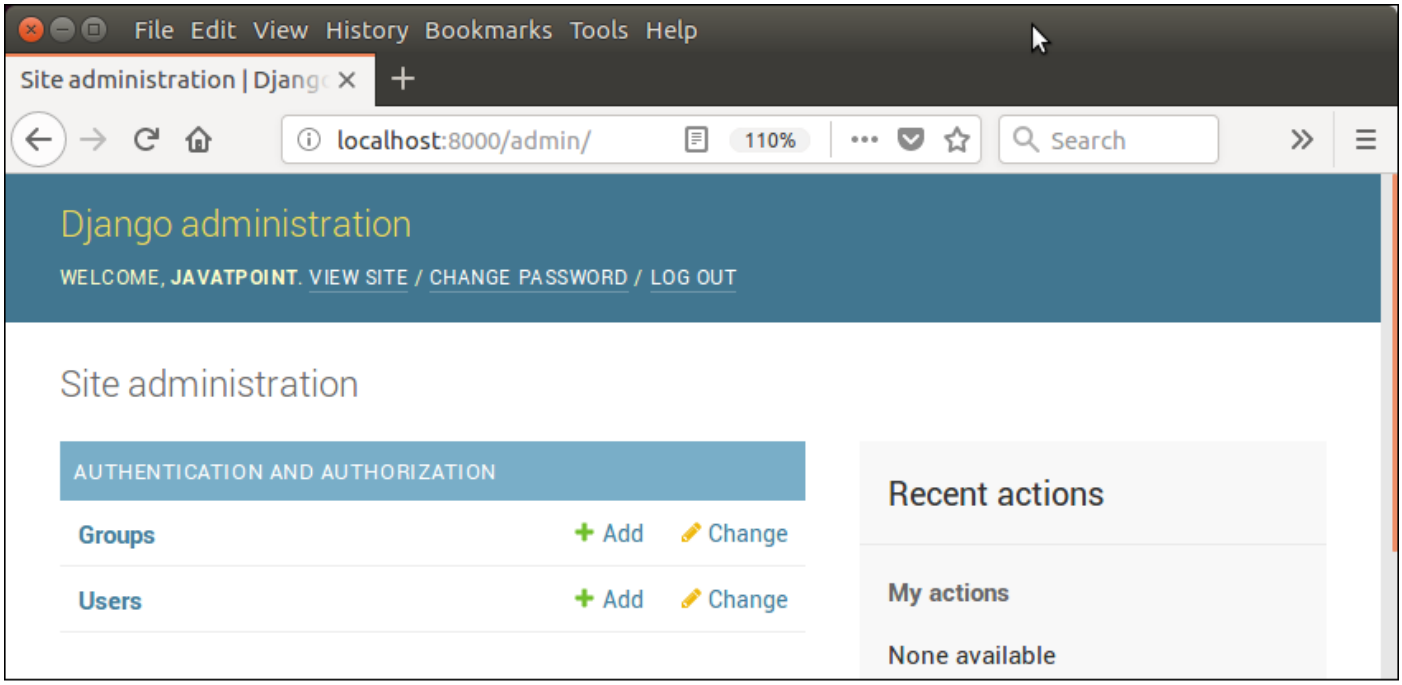
To login, first create admin (super user) user and provide password as we did here:



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.

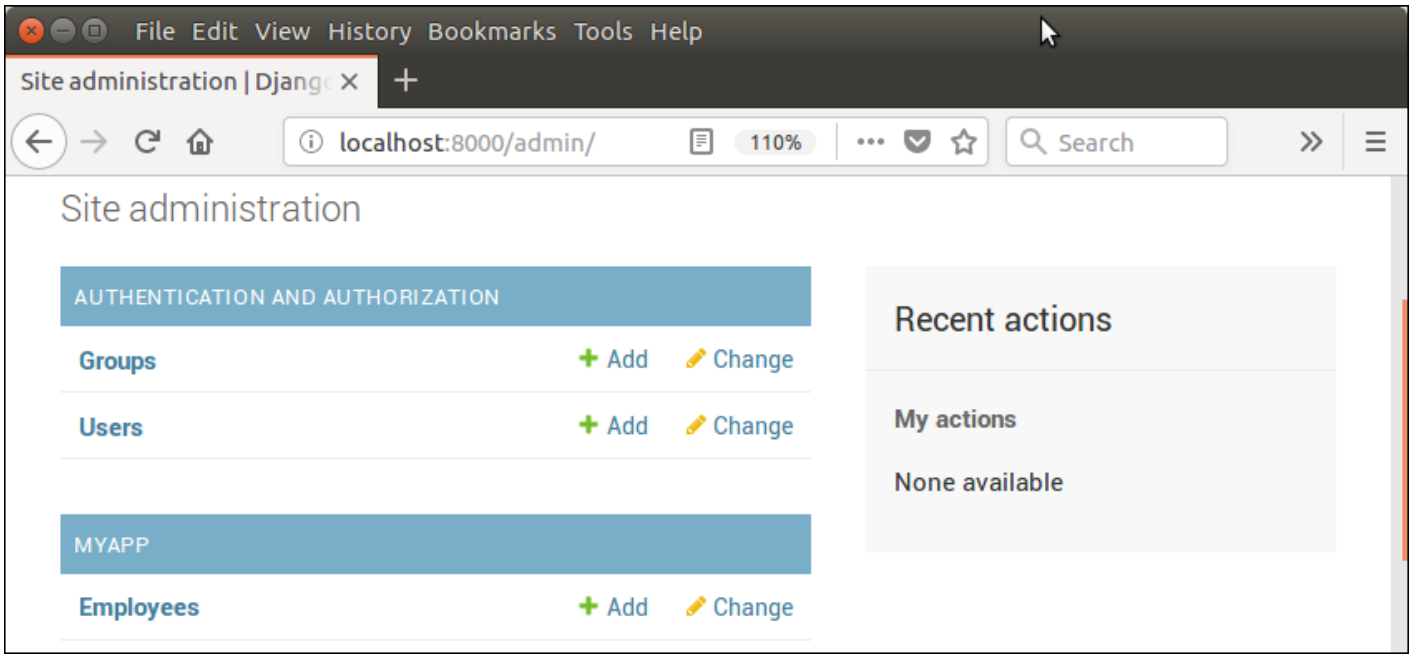
Register Django Model

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

// admin.py

```
from django.contrib import admin
from myapp.models import Employee
admin.site.register(Employee) # Employee is registered
```

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



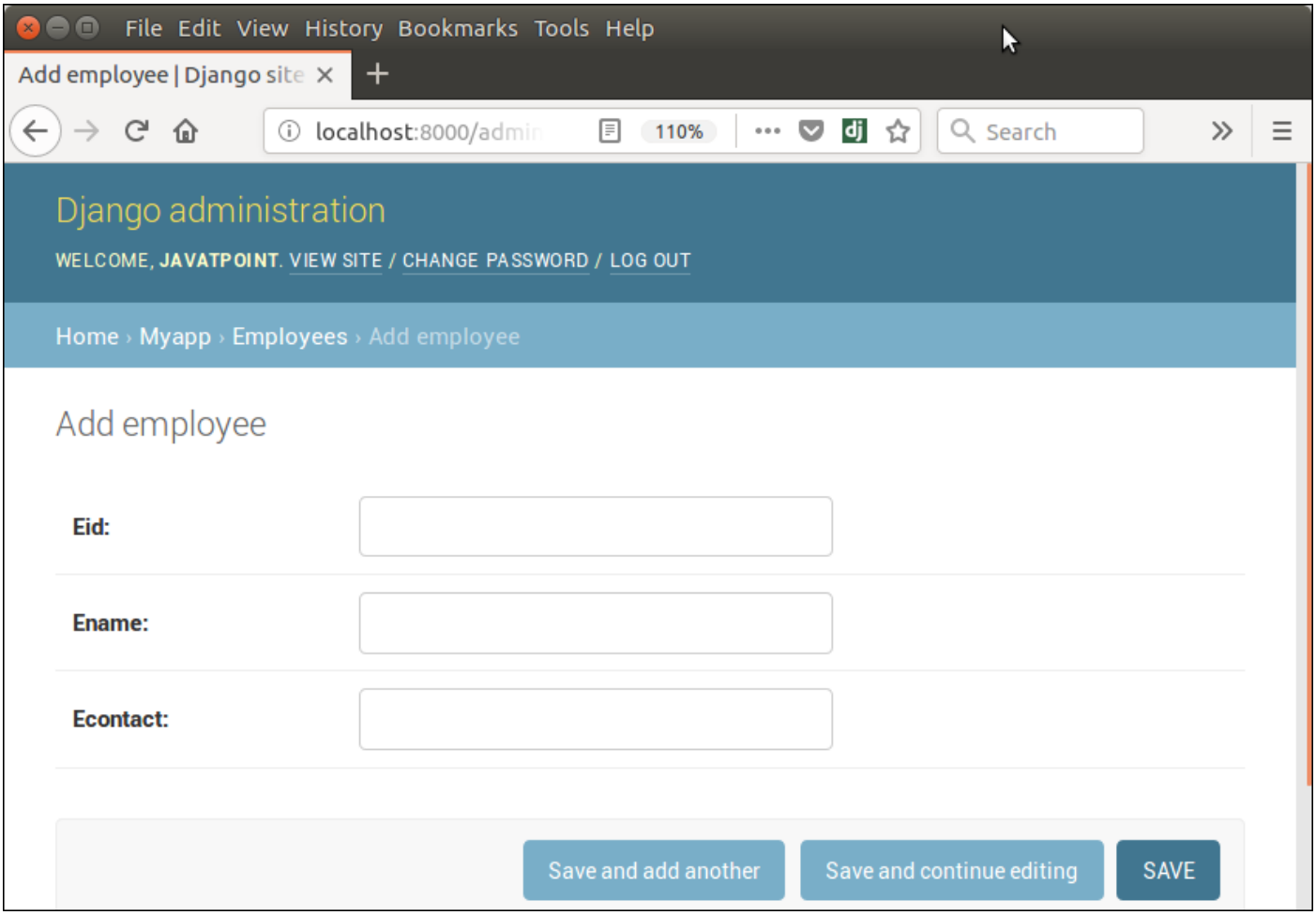
It provides auto generated interface to create new model object. Like, if i click on **add,** it renders a form with all the attributes provided in the model class.

For example, our model class contains the following code.

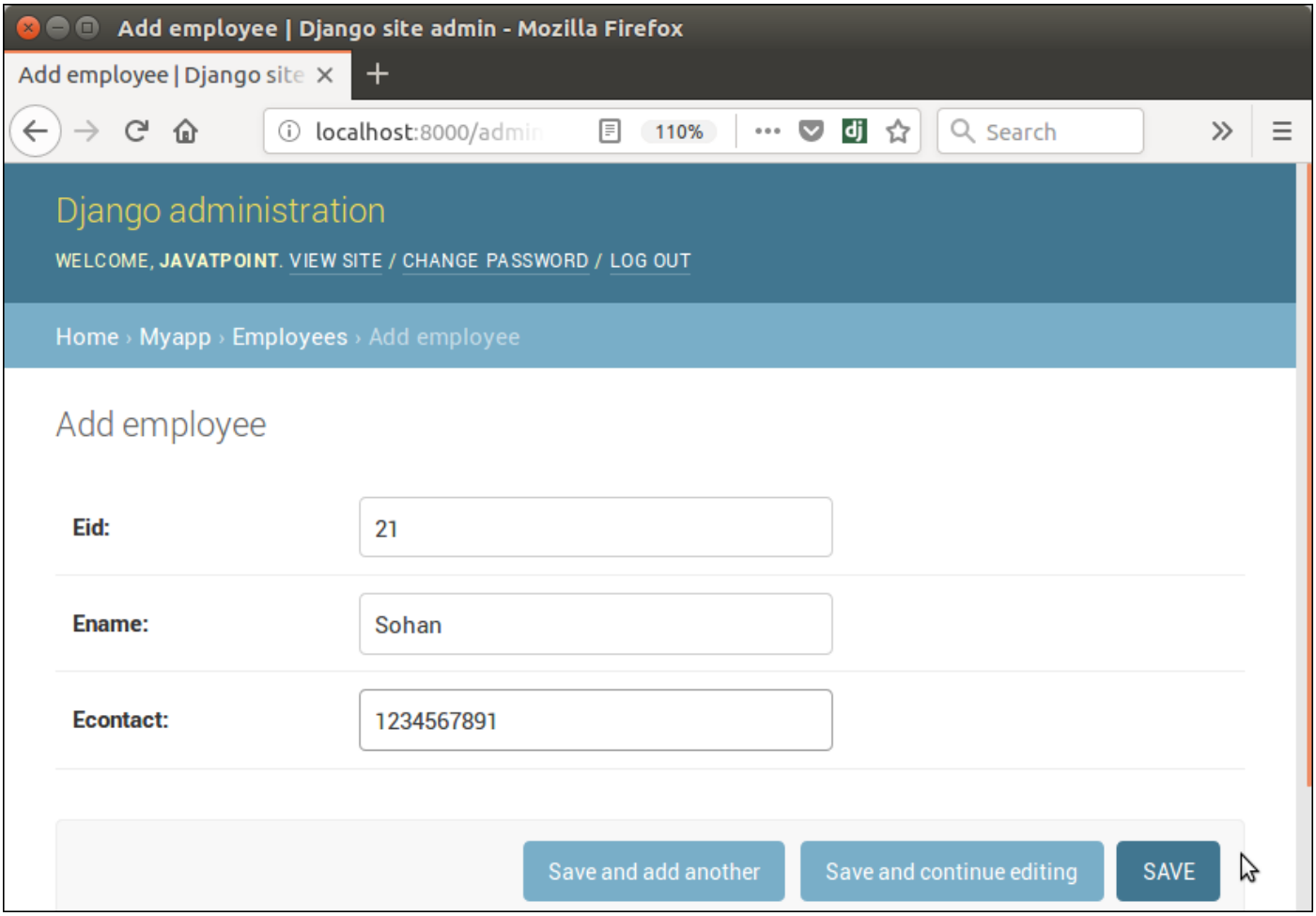
// models.py

```
from django.db import models
class Employee(models.Model):
    eid     = models.CharField(max_length=20)
    ename   = models.CharField(max_length=100)
    econtact = models.CharField(max_length=15)
    class Meta:
        db_table = "employee"
```

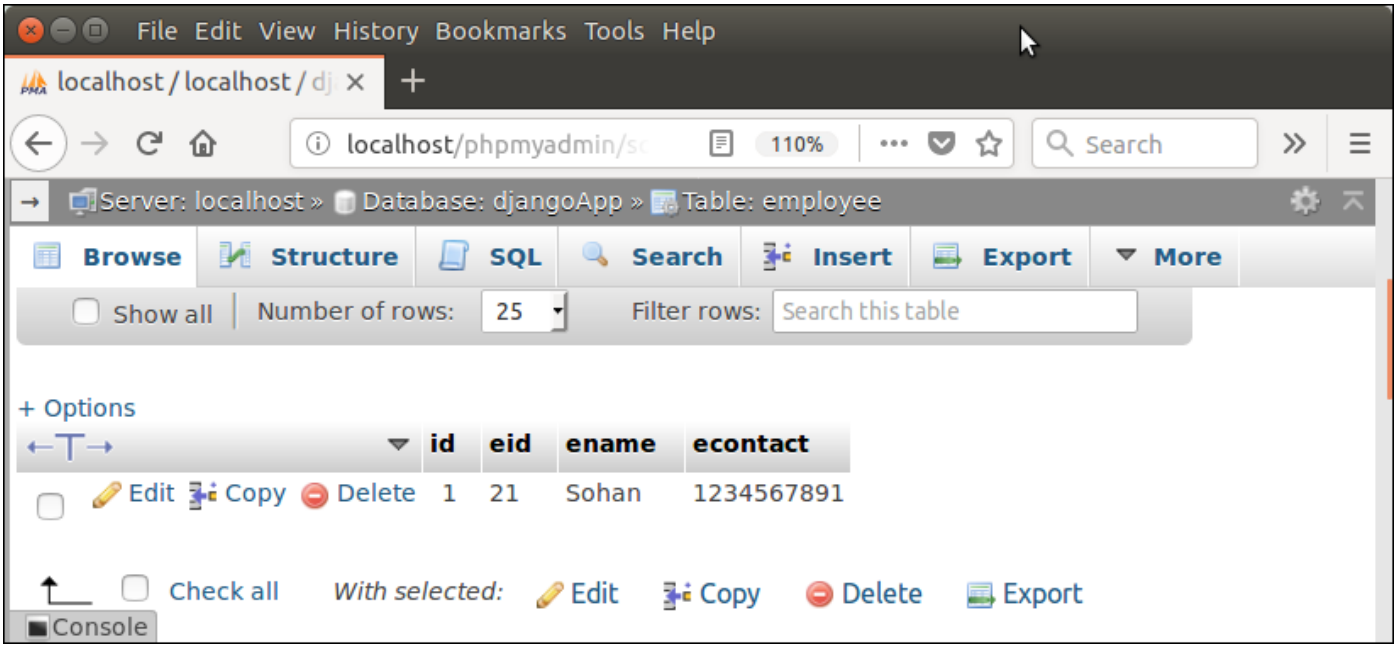
The auto generated form will be based on the model. We don't need to write HTML to create form. The form looks like this:



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.



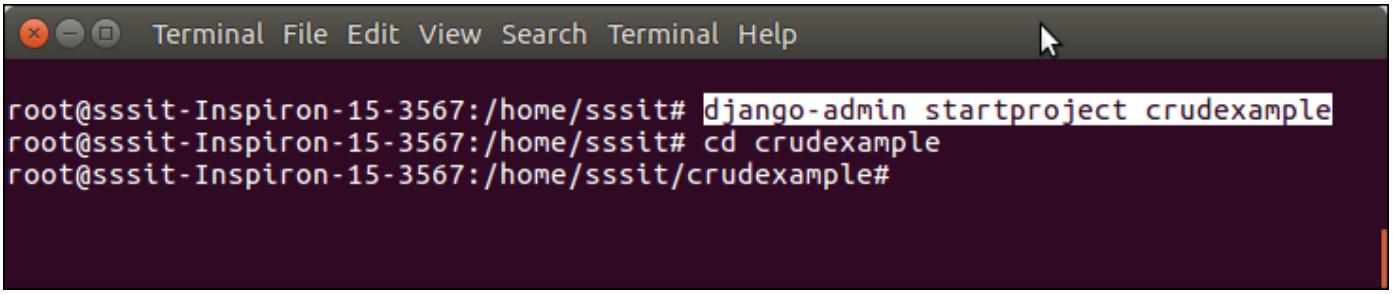
Using this admin dashboard, we can update and delete record also.

Django CRUD (Create Read Update Delete) Example

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

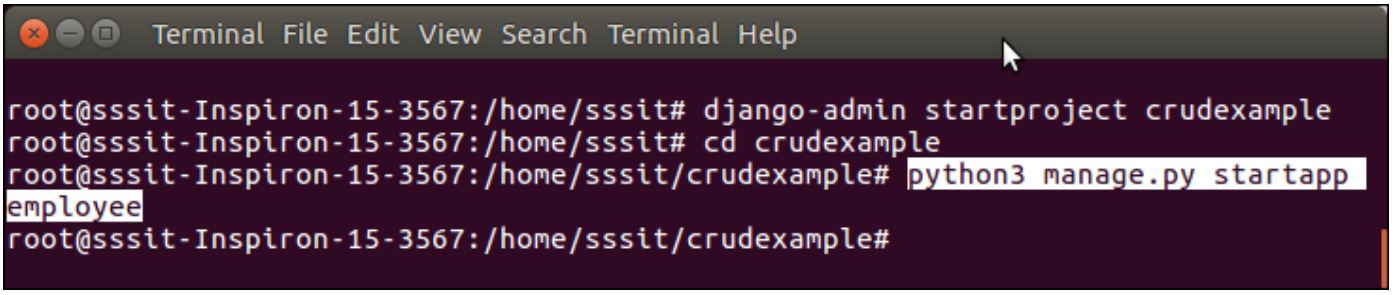
1. Create a Project

$ django-admin startproject crudexample



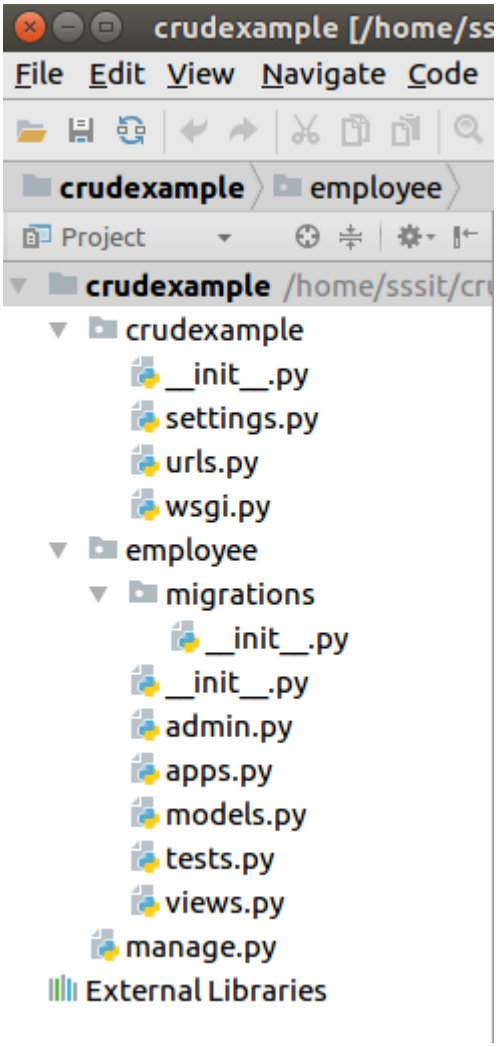
2. Create an App

$ 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

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

5. Create a Model

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

// models.py

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

```
class Employee(models.Model):
    eid = models.CharField(max_length=20)
    ename = models.CharField(max_length=100)
    eemail = models.EmailField()
    econtact = models.CharField(max_length=15)
    class Meta:
        db_table = "employee"
```

6. Create a ModelForm

// forms.py

```
from django import forms
from employee.models import Employee
class EmployeeForm(forms.ModelForm):
    class Meta:
        model = Employee
        fields = "__all__"
```

7. Create View Functions

// views.py

```
from django.shortcuts import render, redirect
from employee.forms import EmployeeForm
from employee.models import Employee
# Create your views here.
def emp(request):
    if request.method == "POST":
        form = EmployeeForm(request.POST)
        if form.is_valid():
            try:
                form.save()
                return redirect('/show')
            except:
                pass
    else:
        form = EmployeeForm()
    return render(request,'index.html',{'form':form})
def show(request):
    employees = Employee.objects.all()
    return render(request,"show.html",{'employees':employees})
def edit(request, id):
    employee = Employee.objects.get(id=id)
    return render(request,'edit.html', {'employee':employee})
def update(request, id):
    employee = Employee.objects.get(id=id)
    form = EmployeeForm(request.POST, instance = employee)
    if form.is_valid():
        form.save()
```

```
        return redirect("/show")
    return render(request, 'edit.html', {'employee': employee})
def destroy(request, id):
    employee = Employee.objects.get(id=id)
    employee.delete()
    return redirect("/show")
```

8. Provide Routing

Provide URL patterns to map with views function.

// urls.py

```
from django.contrib import admin
from django.urls import path
from employee import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('emp', views.emp),
    path('show',views.show),
    path('edit/<int:id>', views.edit),
    path('update/<int:id>', views.update),
    path('delete/<int:id>', views.destroy),
]
```

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

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
    {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
</head>
<body>
<form method="POST" class="post-form" action="/emp">
        {% csrf_token %}
    <div class="container">
<br>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <h3>Enter Details</h3>
    </div>
  </div>
```

```
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Id:</label>
    <div class="col-sm-4">
      {{ form.eid }}
    </div>
  </div>
  <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Name:</label>
    <div class="col-sm-4">
      {{ form.ename }}
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Email:</label>
    <div class="col-sm-4">
      {{ form.eemail }}
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Contact:</label>
    <div class="col-sm-4">
      {{ form.econtact }}
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <button type="submit" class="btn btn-primary">Submit</button>
    </div>
  </div>
    </div>
</form>
</body>
</html>
```

// show.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Employee Records</title>
     {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
</head>
<body>
<table class="table table-striped table-bordered table-sm">
    <thead class="thead-dark">
    <tr>
```

```
        <th>Employee ID</th>
        <th>Employee Name</th>
        <th>Employee Email</th>
        <th>Employee Contact</th>
        <th>Actions</th>
    </tr>
    </thead>
    <tbody>
{% for employee in employees %}
    <tr>
        <td>{{ employee.eid }}</td>
        <td>{{ employee.ename }}</td>
        <td>{{ employee.eemail }}</td>
        <td>{{ employee.econtact }}</td>
        <td>
            <a href="/edit/{{ employee.id }}"><span class="glyphicon glyphicon-
pencil" >Edit</span></a>
            <a href="/delete/{{ employee.id }}">Delete</a>
        </td>
    </tr>
{% endfor %}
    </tbody>
</table>
<br>
<br>
<center><a href="/emp" class="btn btn-primary">Add New Record</a></center>
</body>
</html>
```

// edit.html

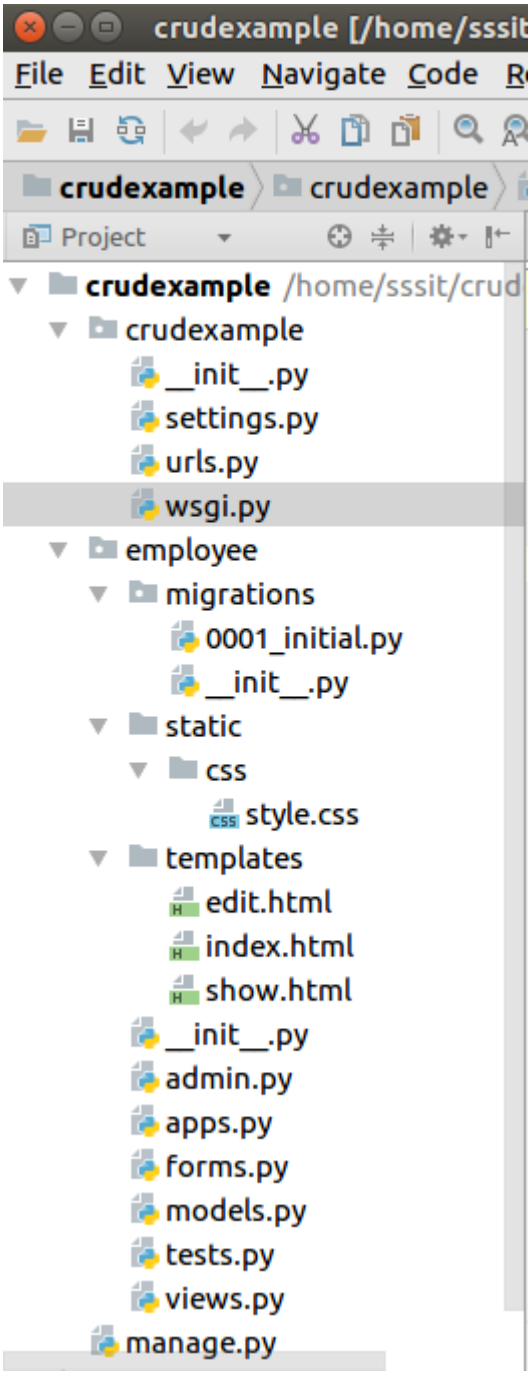
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
    {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
</head>
<body>
<form method="POST" class="post-form" action="/update/{{employee.id}}">
        {% csrf_token %}
    <div class="container">
<br>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <h3>Update Details</h3>
    </div>
```

```
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Id:</label>
    <div class="col-sm-4">
       <input type="text" name="eid" id="id_eid" required maxlength="20" value="{{ employee.eid }
}"/>
    </div>
  </div>
  <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Name:</label>
    <div class="col-sm-4">
       <input type="text" name="ename" id="id_ename" required maxlength="100" value="{{ employ
ee.ename }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Email:</label>
    <div class="col-sm-4">
       <input type="email" name="eemail" id="id_eemail" required maxlength="254" value="{{ emplo
yee.eemail }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Contact:</label>
    <div class="col-sm-4">
       <input type="text" name="econtact" id="id_econtact" required maxlength="15" value="{{ empl
oyee.econtact }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <button type="submit" class="btn btn-success">Update</button>
    </div>
  </div>
    </div>
</form>
</body>
</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.

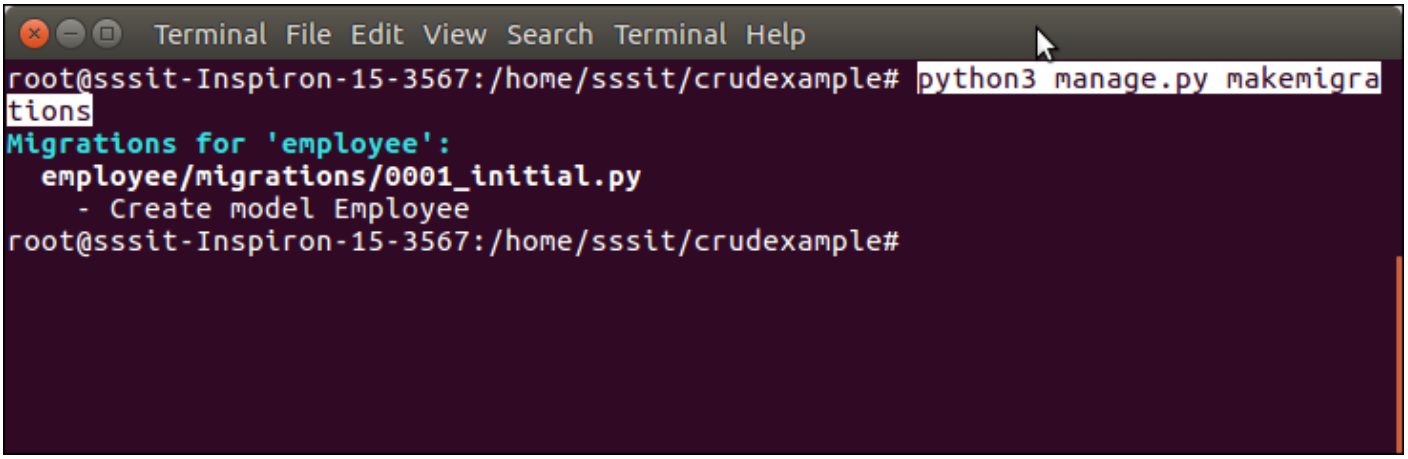
11. Project Structure



12. Create Migrations

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

$ python3 manage.py makemigrations



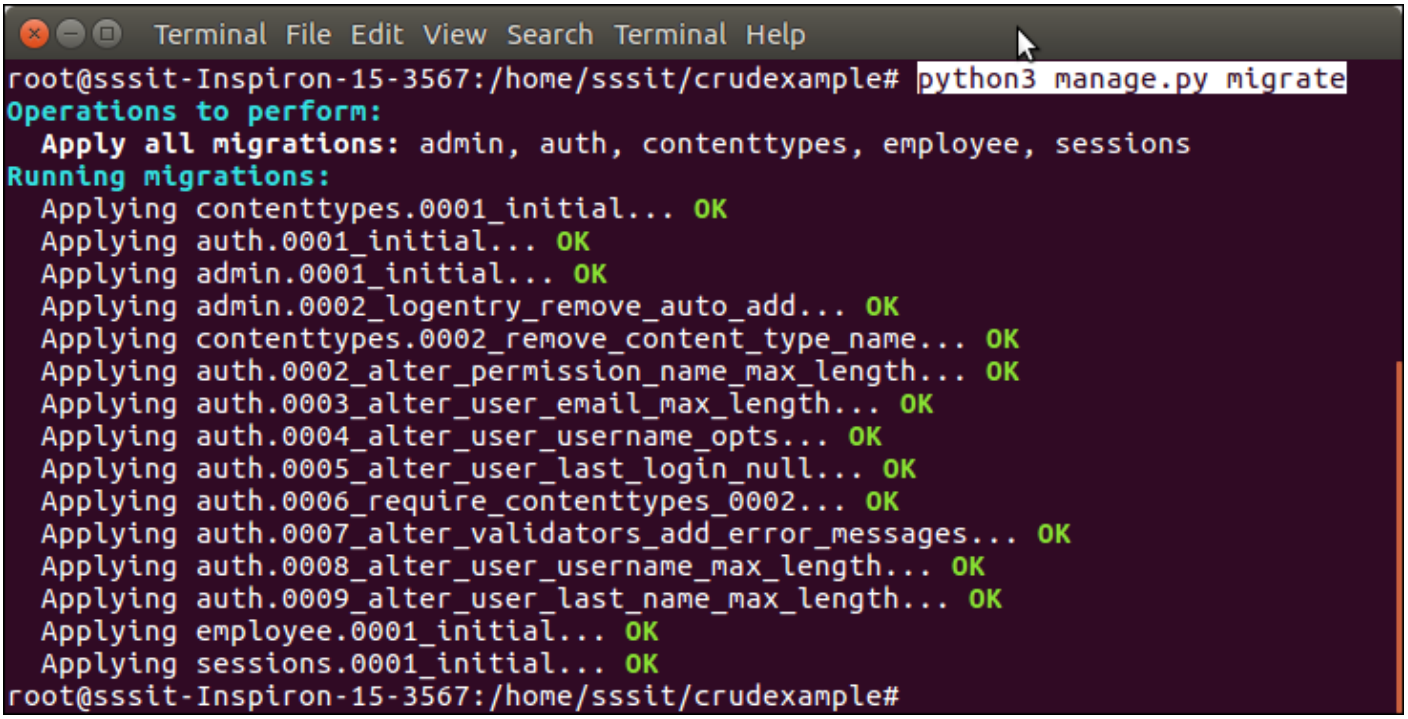
After migrations, execute one more command to reflect the migration into the database. But before it, mention name of app (employee) in INSTALLED_APPS of settings.py file.

// settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'employee'
]
```

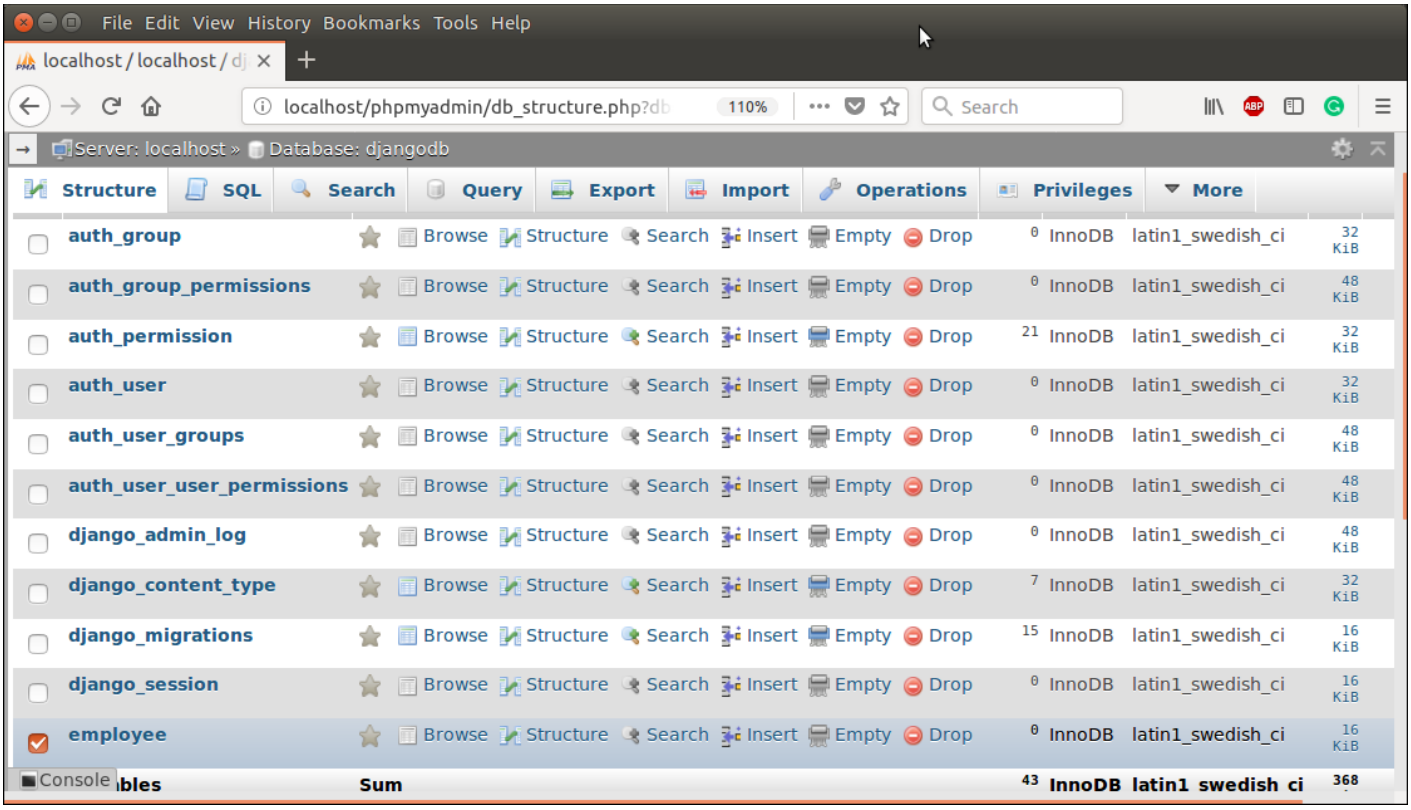
Run the command to migrate the migrations.

$ 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.

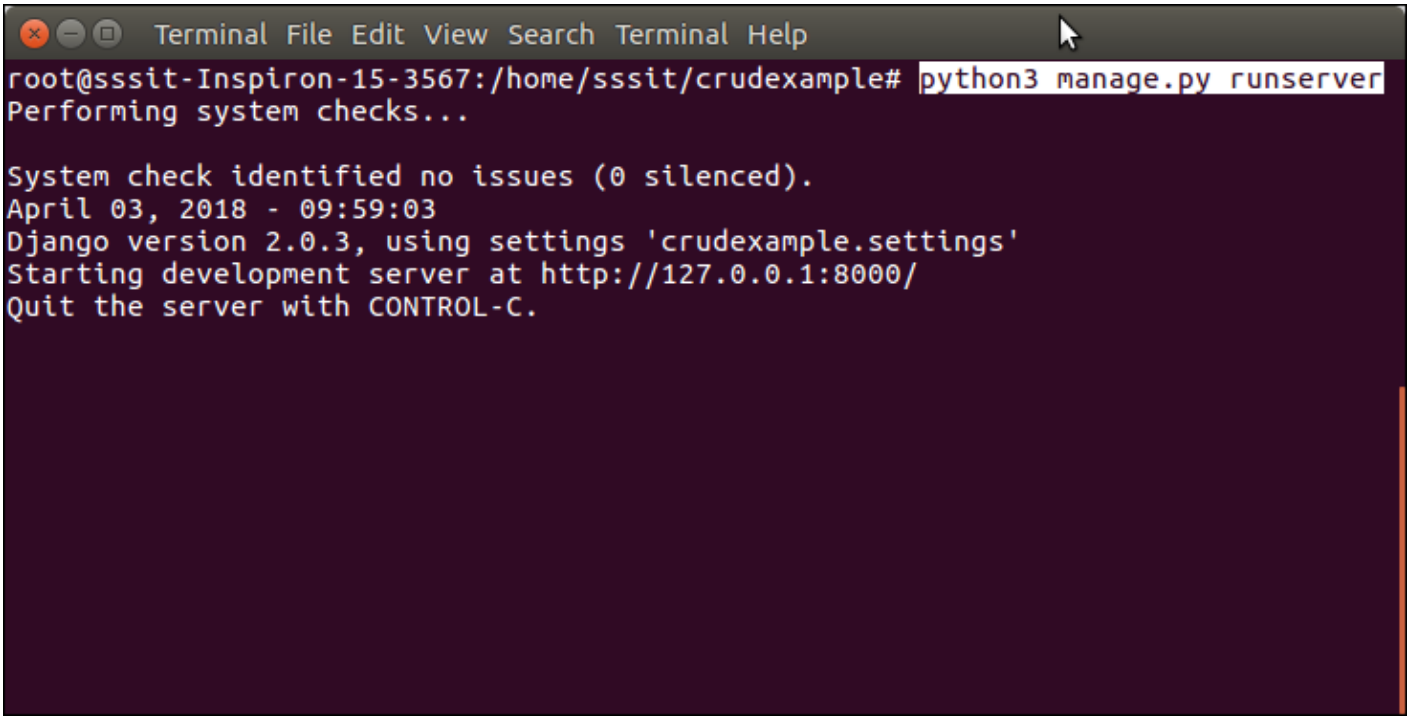
See list of tables created after migrate command.



Run Server

To run server use the following command.

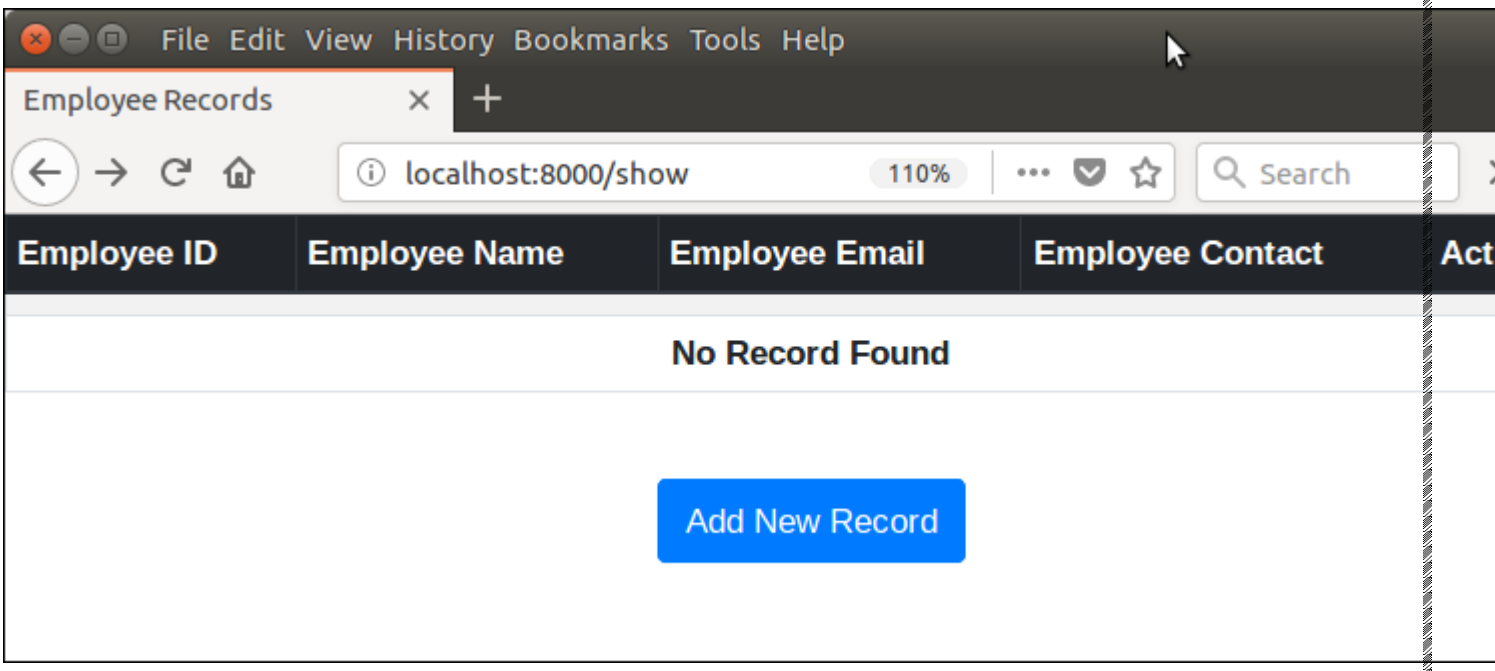
$ python3 manage.py runserver



Access to the Browser

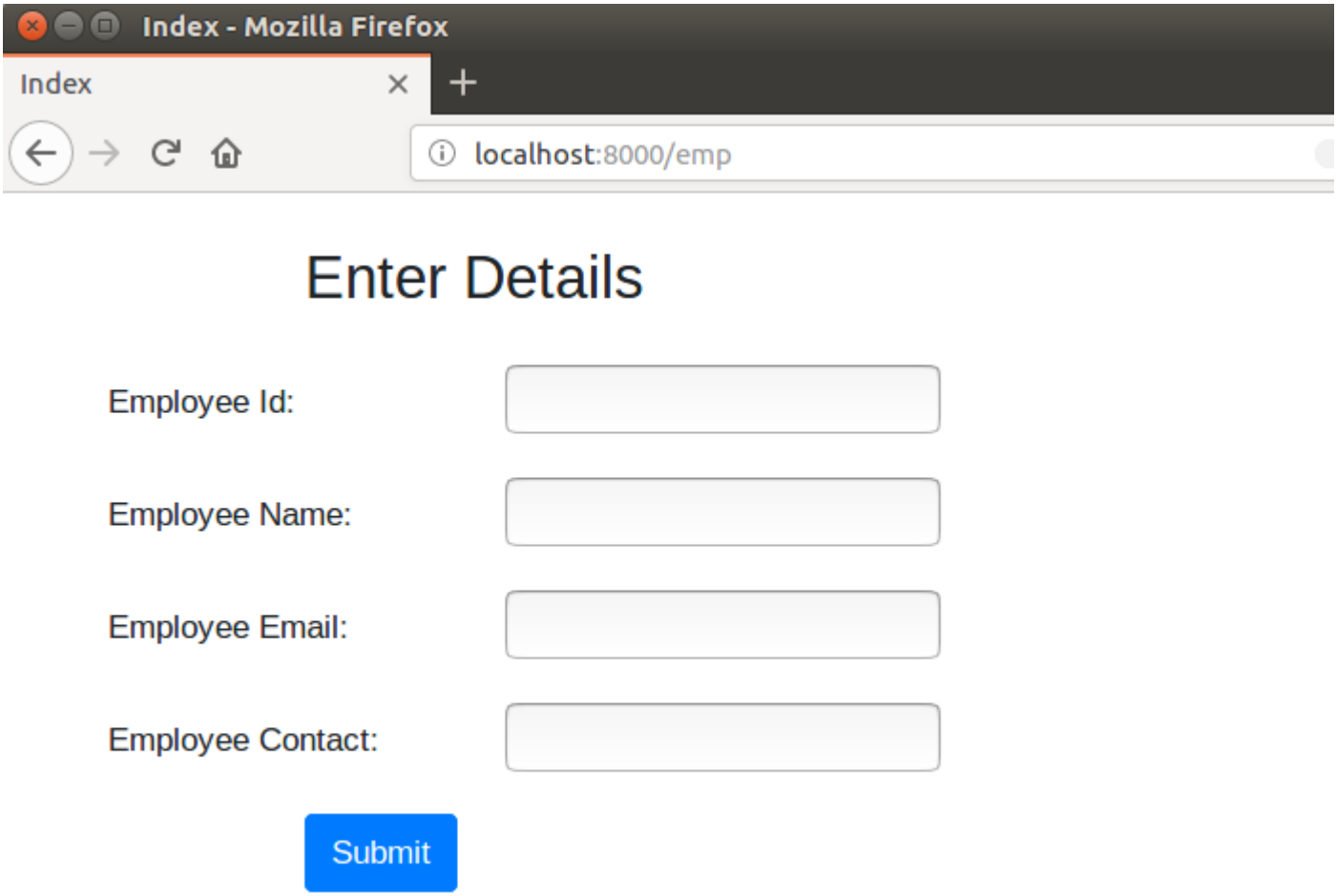
Access the application by entering **localhost:8000/show**, it will show all the available employee records.

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

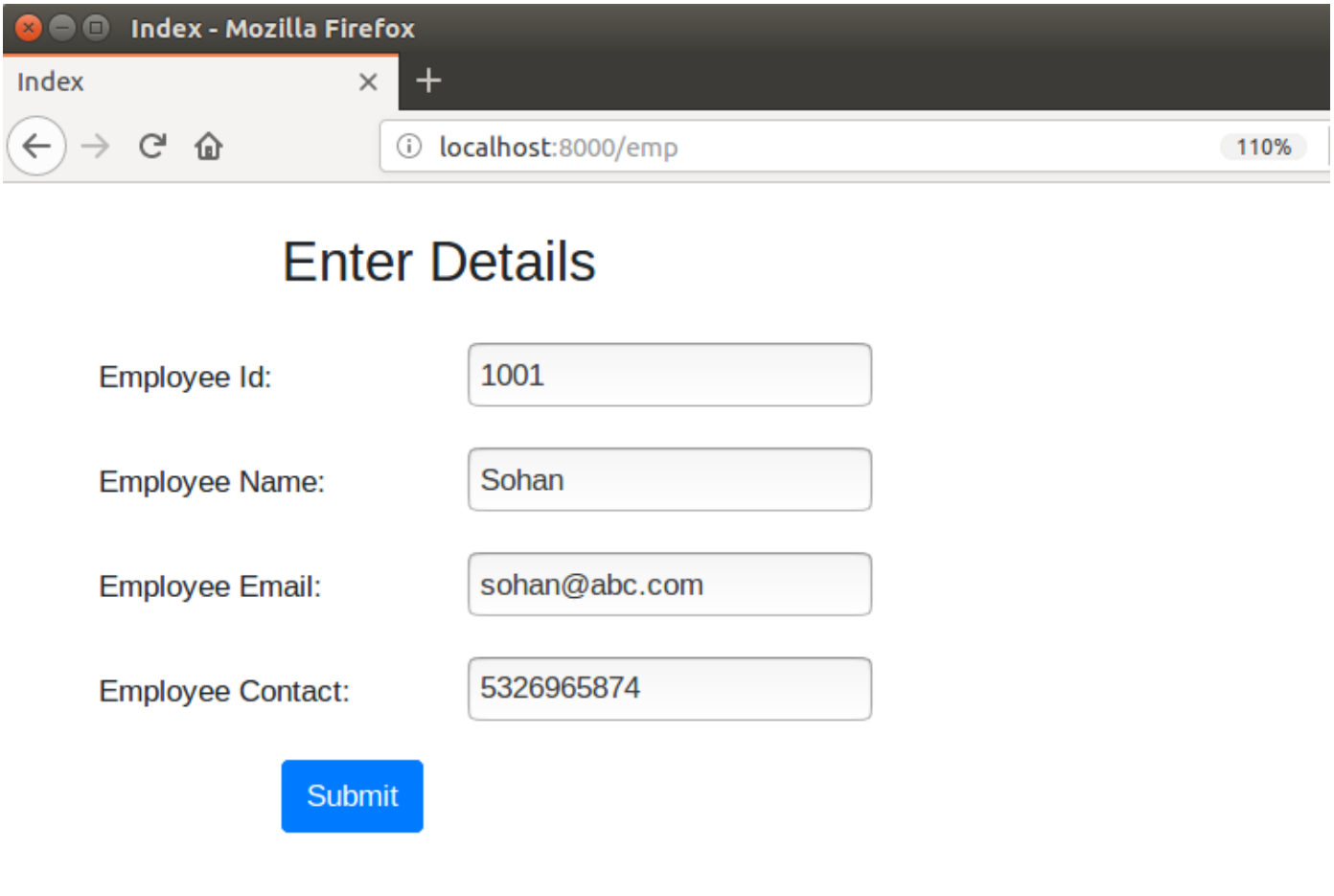


Adding Record

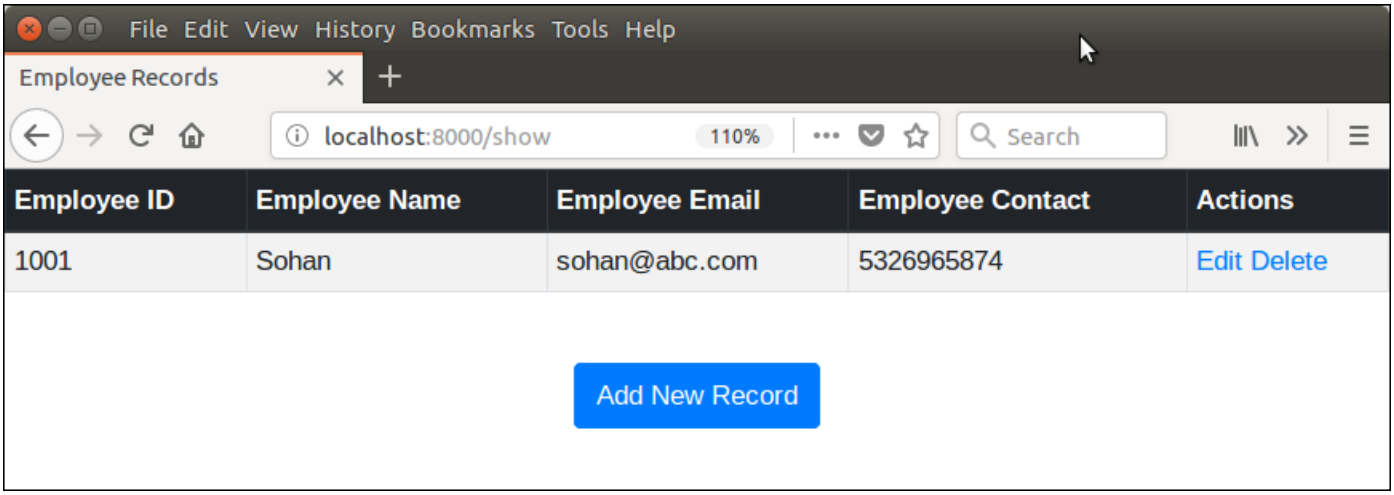
Click on the **Add New Record** button and fill the details. See the example.



Filling the details.

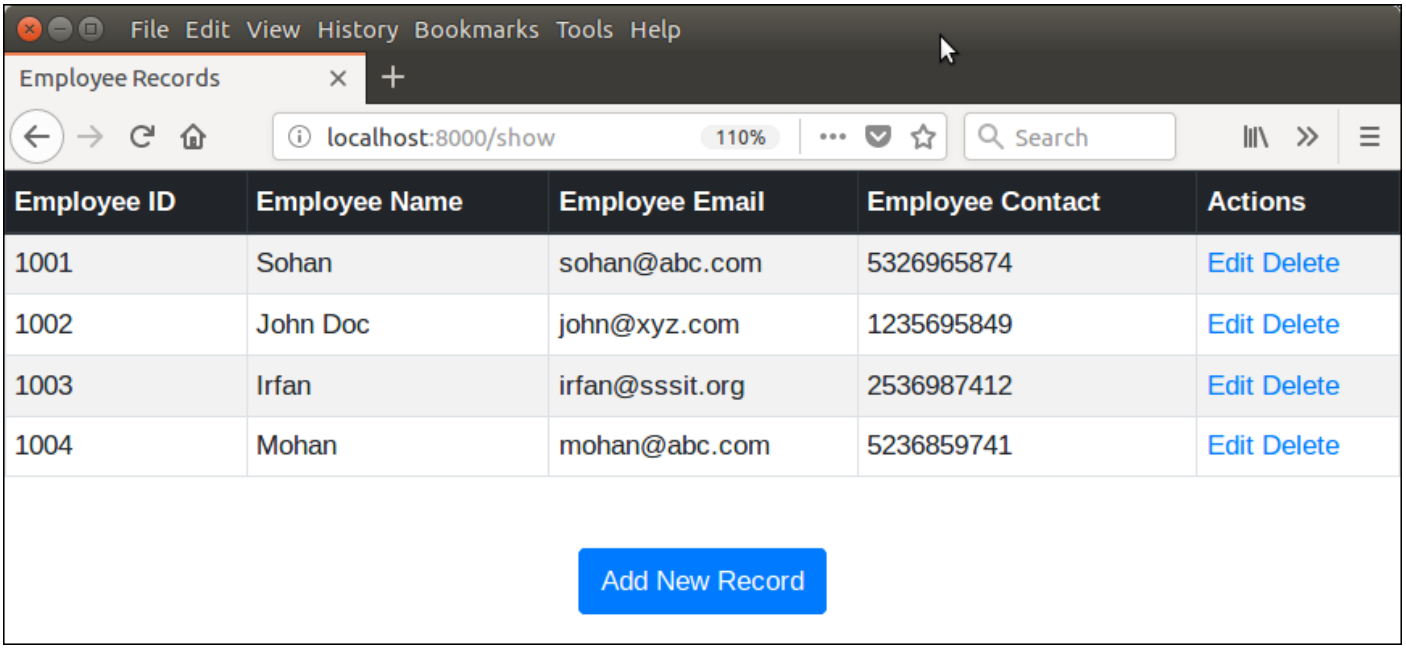


Submit the record and see, after submitting it shows the saved record.



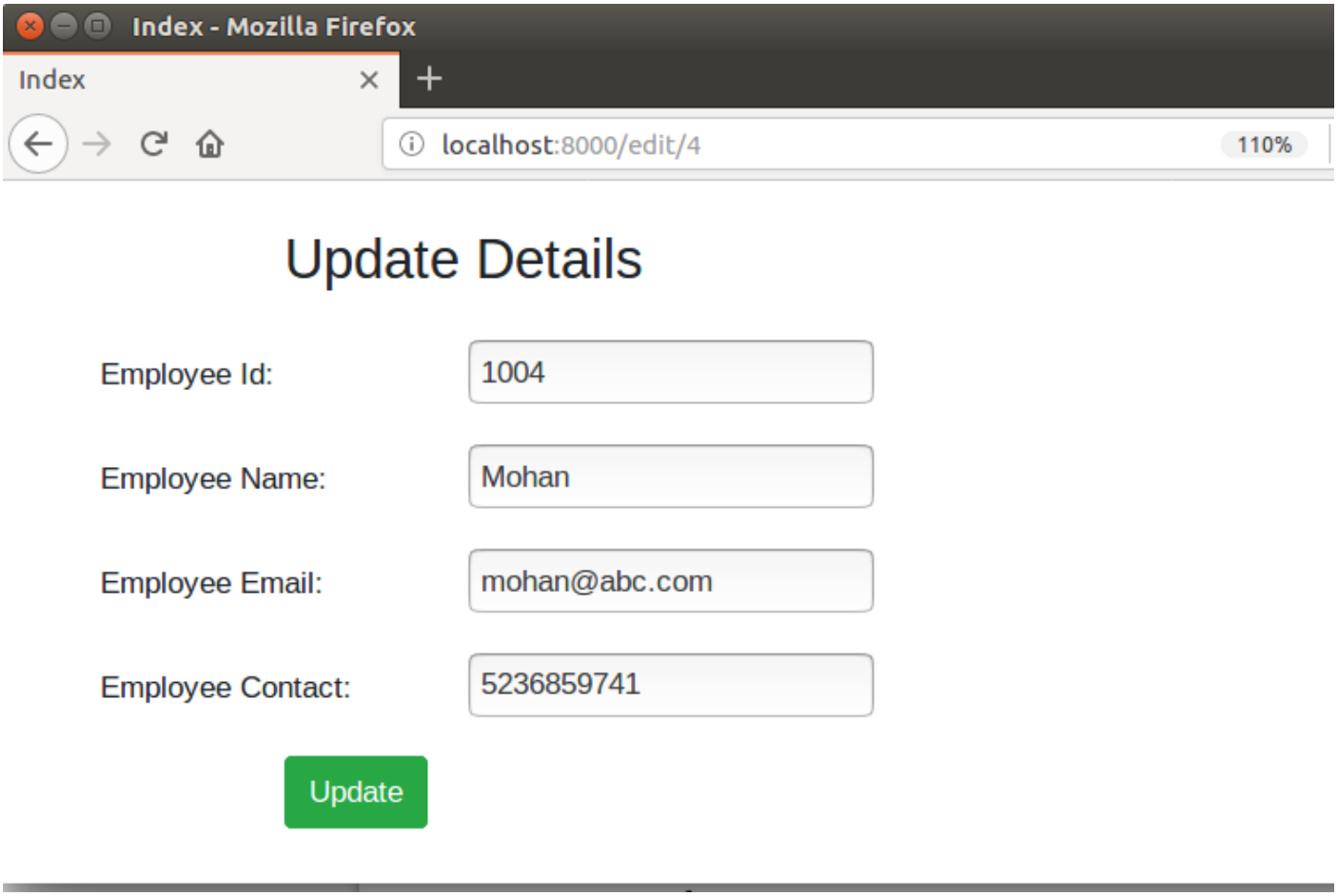
This section also allows, update and delete records from the **actions** column.

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

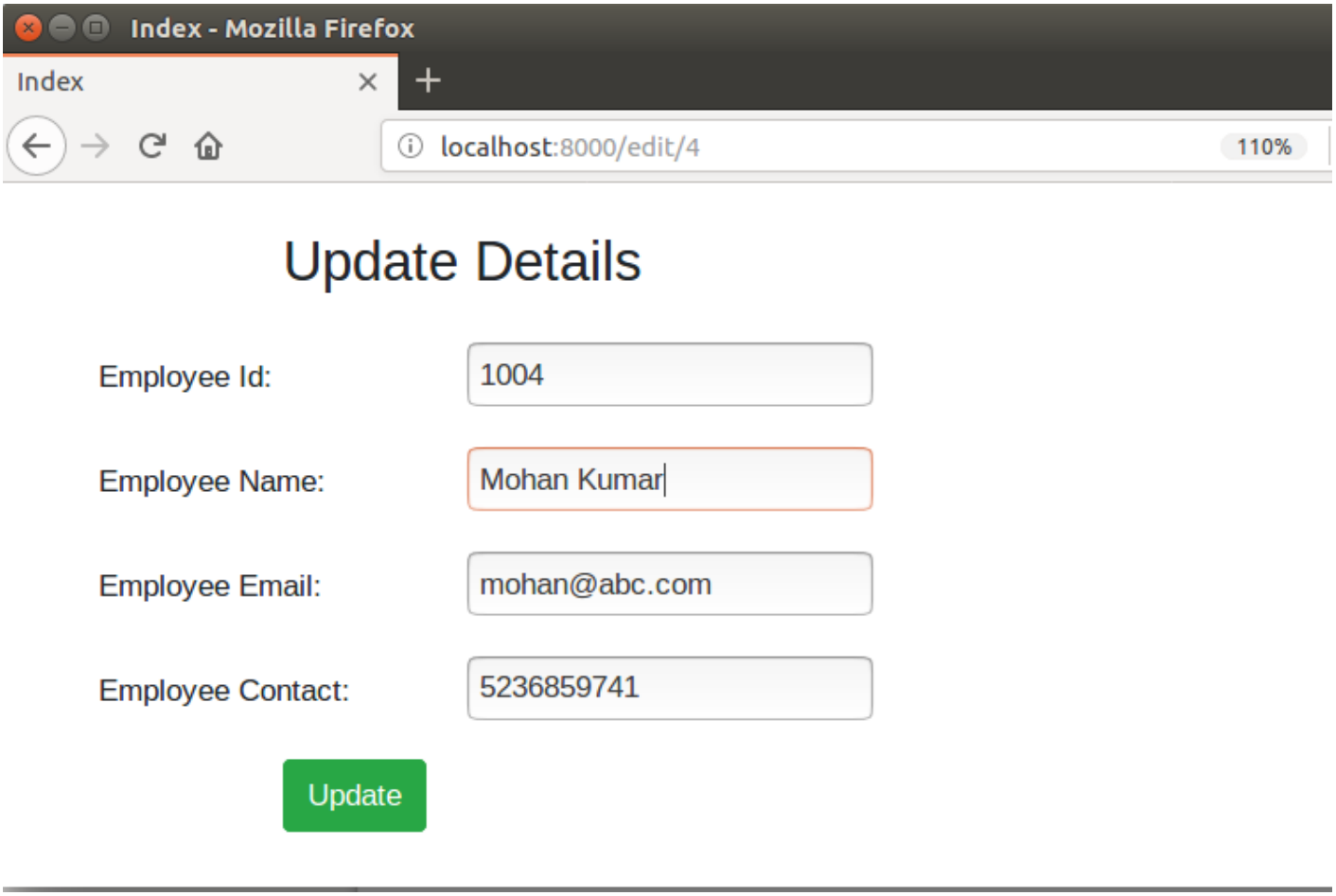


Update Record

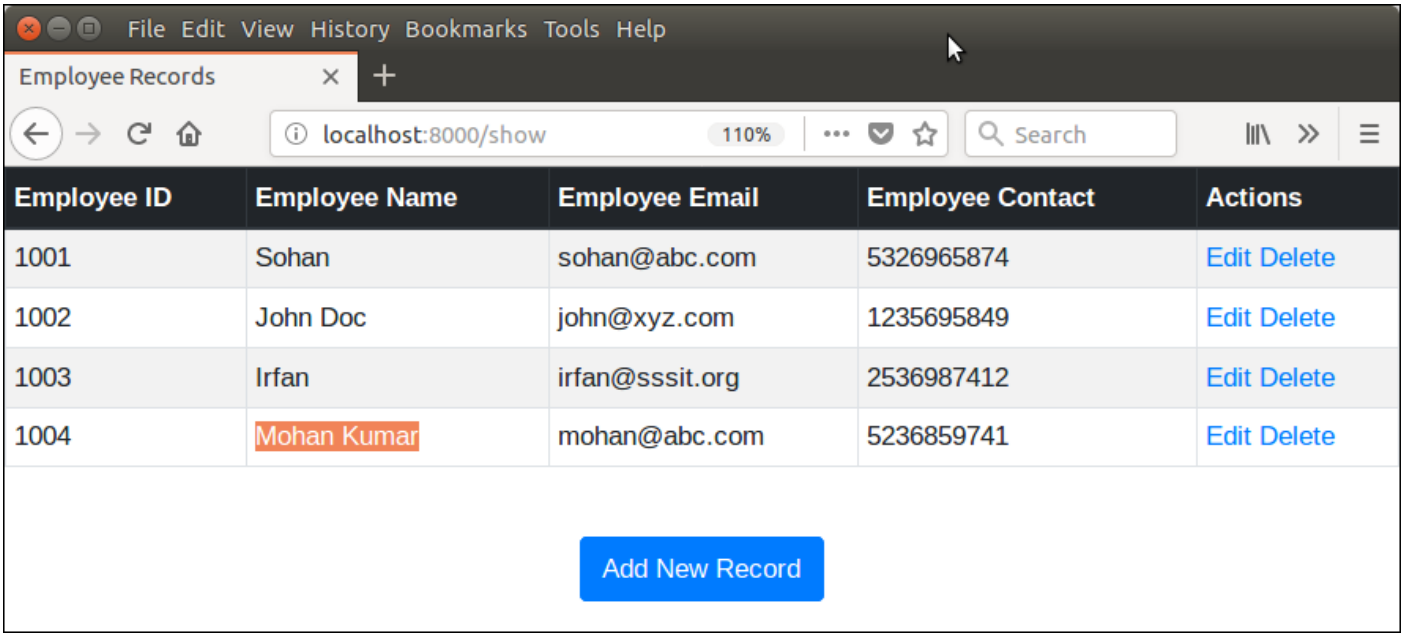
Lets update the record of **Mohan** by clicking on **edit** button. It will display record of Mohan in edit mode.



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



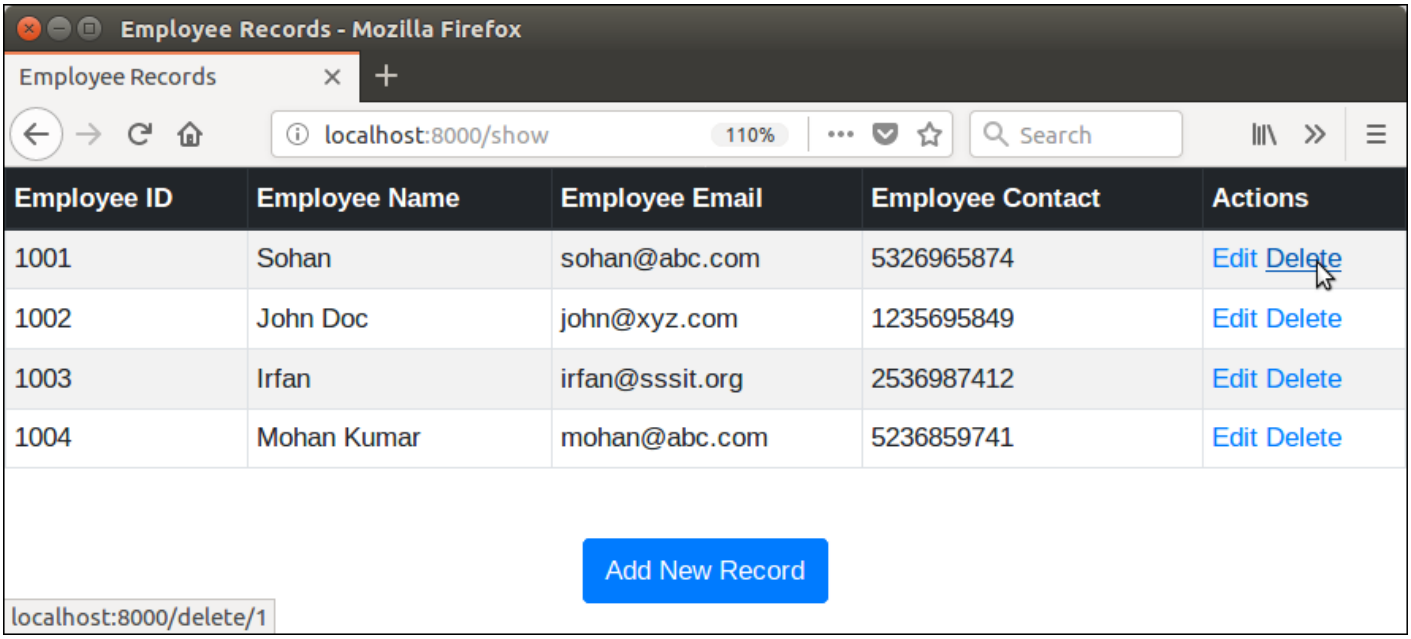
Click on update button and it redirects to the following page. See name is updated.



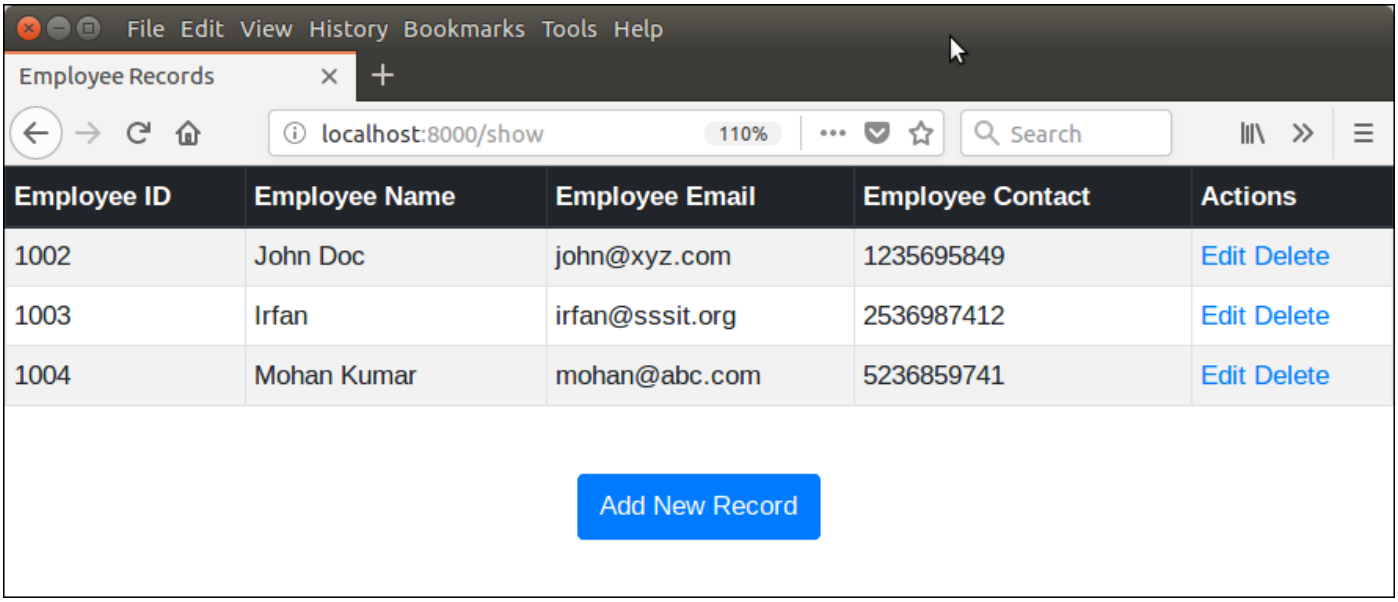
Same like, we can delete records too, by clicking the **delete** link.

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.

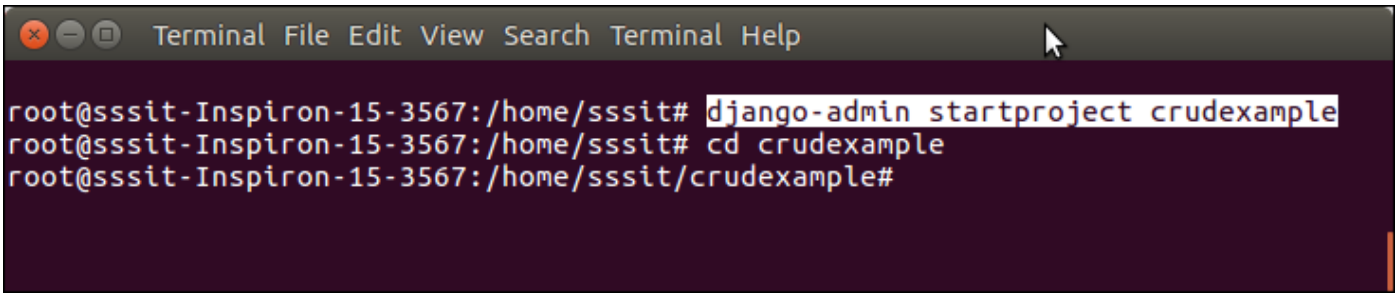
This complete project can be downloaded here.

Django CRUD (Create Read Update Delete) Example

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

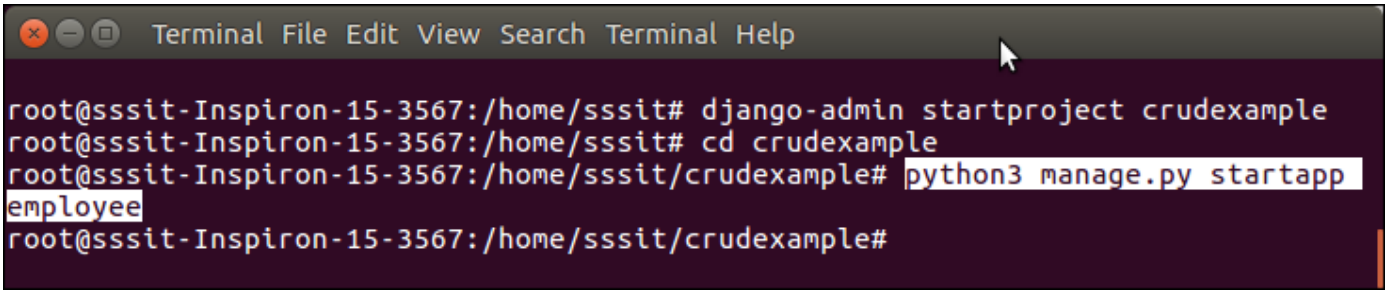
1. Create a Project

$ django-admin startproject crudexample



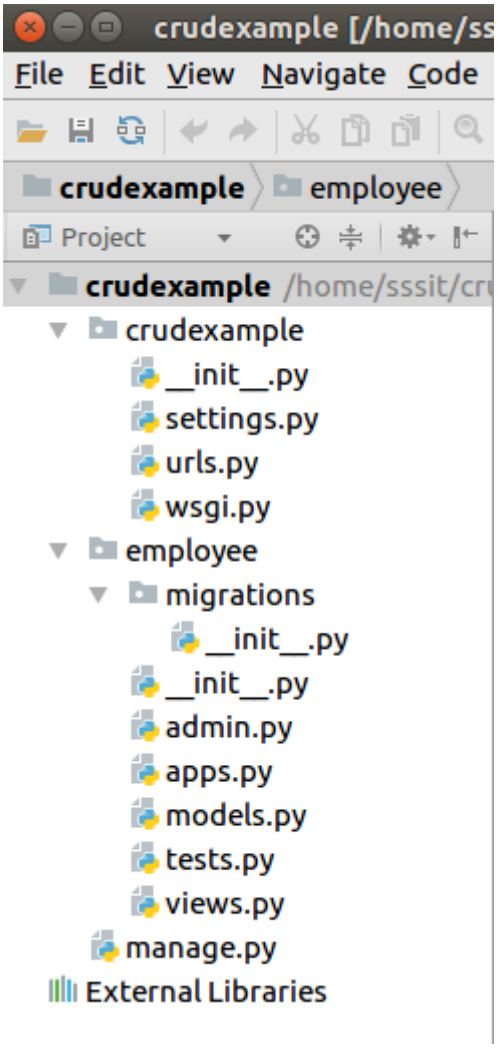
2. Create an App

$ 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

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

5. Create a Model

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

// models.py

```
from django.db import models
class Employee(models.Model):
    eid = models.CharField(max_length=20)
    ename = models.CharField(max_length=100)
    eemail = models.EmailField()
    econtact = models.CharField(max_length=15)
    class Meta:
        db_table = "employee"
```

6. Create a ModelForm

// forms.py

```
from django import forms
from employee.models import Employee
class EmployeeForm(forms.ModelForm):
    class Meta:
        model = Employee
        fields = "__all__"
```

7. Create View Functions

// views.py

```
from django.shortcuts import render, redirect
from employee.forms import EmployeeForm
from employee.models import Employee
# Create your views here.
def emp(request):
    if request.method == "POST":
        form = EmployeeForm(request.POST)
        if form.is_valid():
            try:
                form.save()
                return redirect('/show')
            except:
                pass
    else:
        form = EmployeeForm()
    return render(request,'index.html',{'form':form})
def show(request):
    employees = Employee.objects.all()
    return render(request,"show.html",{'employees':employees})
def edit(request, id):
    employee = Employee.objects.get(id=id)
    return render(request,'edit.html', {'employee':employee})
def update(request, id):
    employee = Employee.objects.get(id=id)
    form = EmployeeForm(request.POST, instance = employee)
    if form.is_valid():
```

```
        form.save()
        return redirect("/show")
    return render(request, 'edit.html', {'employee': employee})
def destroy(request, id):
    employee = Employee.objects.get(id=id)
    employee.delete()
    return redirect("/show")
```

8. Provide Routing

Provide URL patterns to map with views function.

// urls.py

```
from django.contrib import admin
from django.urls import path
from employee import views
urlpatterns = [
    path('admin/', admin.site.urls),
    path('emp', views.emp),
    path('show',views.show),
    path('edit/<int:id>', views.edit),
    path('update/<int:id>', views.update),
    path('delete/<int:id>', views.destroy),
]
```

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

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
    {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
</head>
<body>
<form method="POST" class="post-form" action="/emp">
        {% csrf_token %}
    <div class="container">
<br>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <h3>Enter Details</h3>
    </div>
```

```
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Id:</label>
    <div class="col-sm-4">
      {{ form.eid }}
    </div>
  </div>
  <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Name:</label>
    <div class="col-sm-4">
      {{ form.ename }}
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Email:</label>
    <div class="col-sm-4">
      {{ form.eemail }}
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Contact:</label>
    <div class="col-sm-4">
      {{ form.econtact }}
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <button type="submit" class="btn btn-primary">Submit</button>
    </div>
  </div>
    </div>
</form>
</body>
</html>
```

// show.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Employee Records</title>
     {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
</head>
<body>
<table class="table table-striped table-bordered table-sm">
    <thead class="thead-dark">
```

```
    <tr>
      <th>Employee ID</th>
      <th>Employee Name</th>
      <th>Employee Email</th>
      <th>Employee Contact</th>
      <th>Actions</th>
    </tr>
    </thead>
    <tbody>
{% for employee in employees %}
    <tr>
      <td>{{ employee.eid }}</td>
      <td>{{ employee.ename }}</td>
      <td>{{ employee.eemail }}</td>
      <td>{{ employee.econtact }}</td>
      <td>
        <a href="/edit/{{ employee.id }}"><span class="glyphicon glyphicon-
pencil" >Edit</span></a>
        <a href="/delete/{{ employee.id }}">Delete</a>
      </td>
    </tr>
{% endfor %}
    </tbody>
</table>
<br>
<br>
<center><a href="/emp" class="btn btn-primary">Add New Record</a></center>
</body>
</html>
```

// edit.html

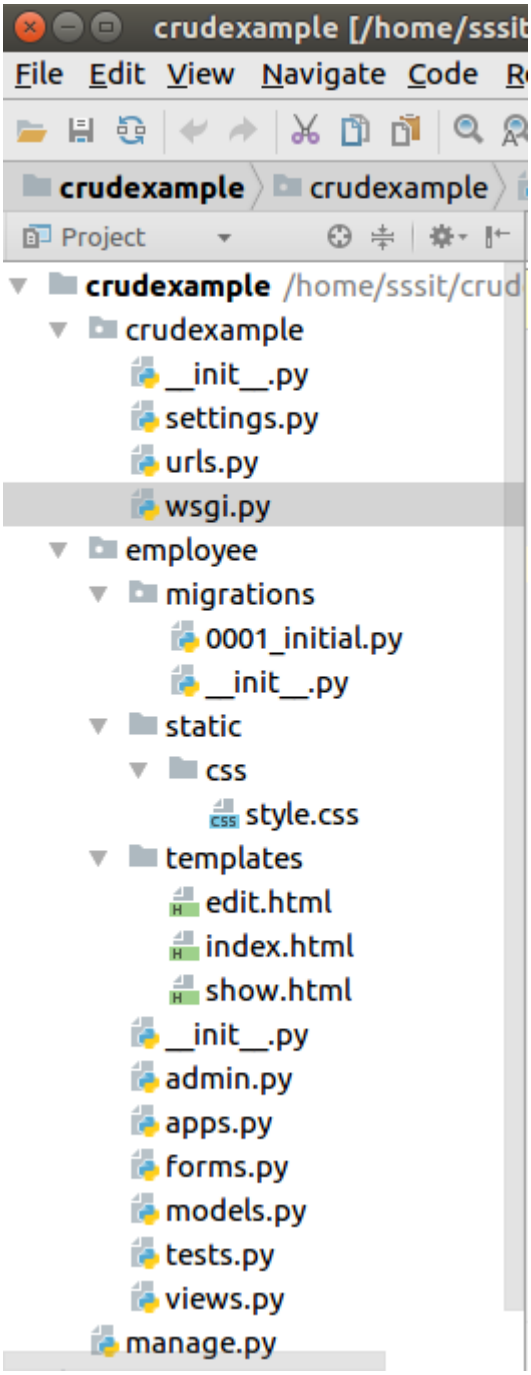
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
    {% load staticfiles %}
    <link rel="stylesheet" href="{% static 'css/style.css' %}"/>
</head>
<body>
<form method="POST" class="post-form" action="/update/{{employee.id}}">
        {% csrf_token %}
    <div class="container">
<br>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <h3>Update Details</h3>
```

```
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Id:</label>
    <div class="col-sm-4">
       <input type="text" name="eid" id="id_eid" required maxlength="20" value="{{ employee.eid }
}"/>
    </div>
  </div>
  <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Name:</label>
    <div class="col-sm-4">
       <input type="text" name="ename" id="id_ename" required maxlength="100" value="{{ employ
ee.ename }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Email:</label>
    <div class="col-sm-4">
       <input type="email" name="eemail" id="id_eemail" required maxlength="254" value="{{ emplo
yee.eemail }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-2 col-form-label">Employee Contact:</label>
    <div class="col-sm-4">
       <input type="text" name="econtact" id="id_econtact" required maxlength="15" value="{{ empl
oyee.econtact }}" />
    </div>
  </div>
    <div class="form-group row">
    <label class="col-sm-1 col-form-label"></label>
    <div class="col-sm-4">
    <button type="submit" class="btn btn-success">Update</button>
    </div>
  </div>
    </div>
</form>
</body>
</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.

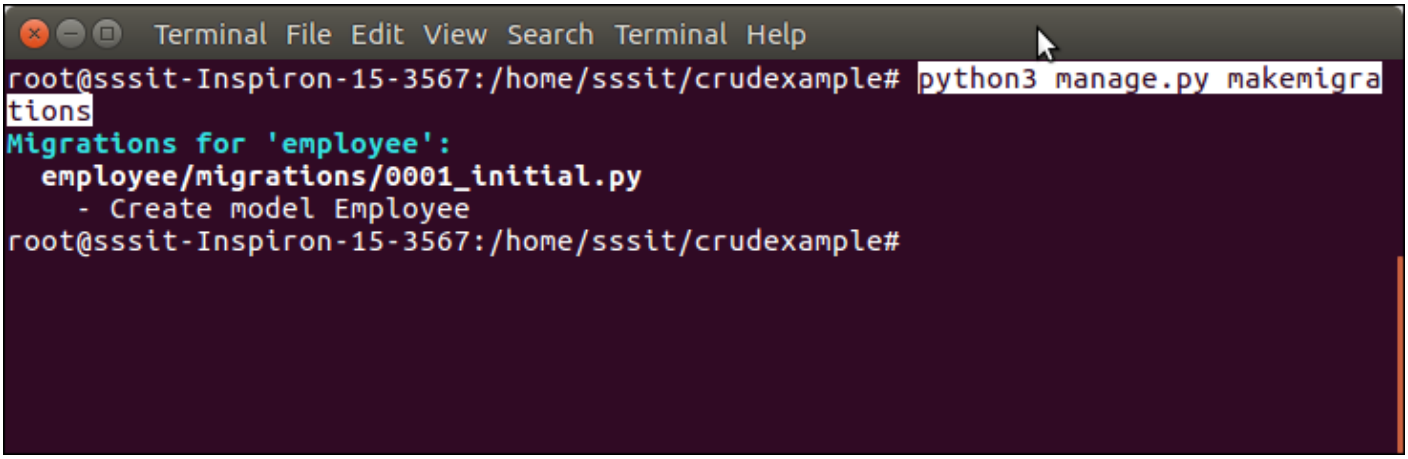
11. Project Structure



12. Create Migrations

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

$ python3 manage.py makemigrations



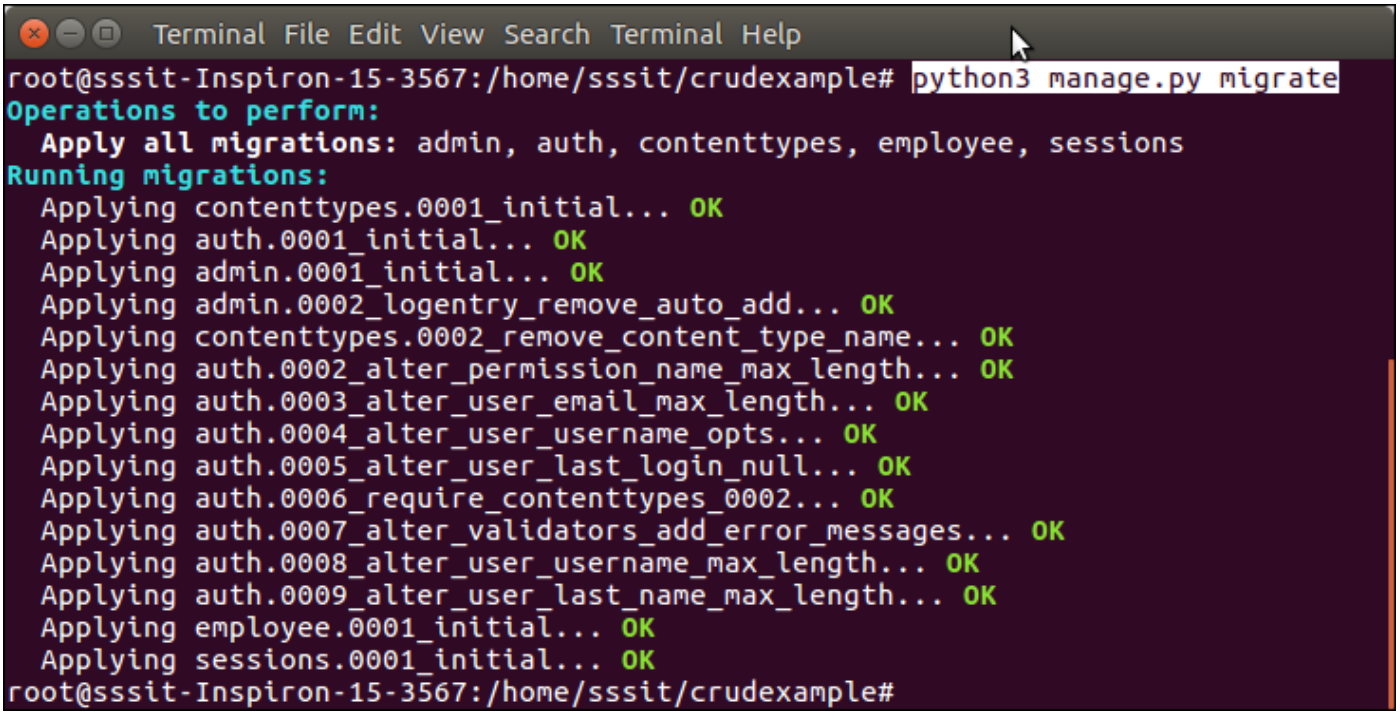
After migrations, execute one more command to reflect the migration into the database. But before it, mention name of app (employee) in INSTALLED_APPS of settings.py file.

// settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'employee'
]
```

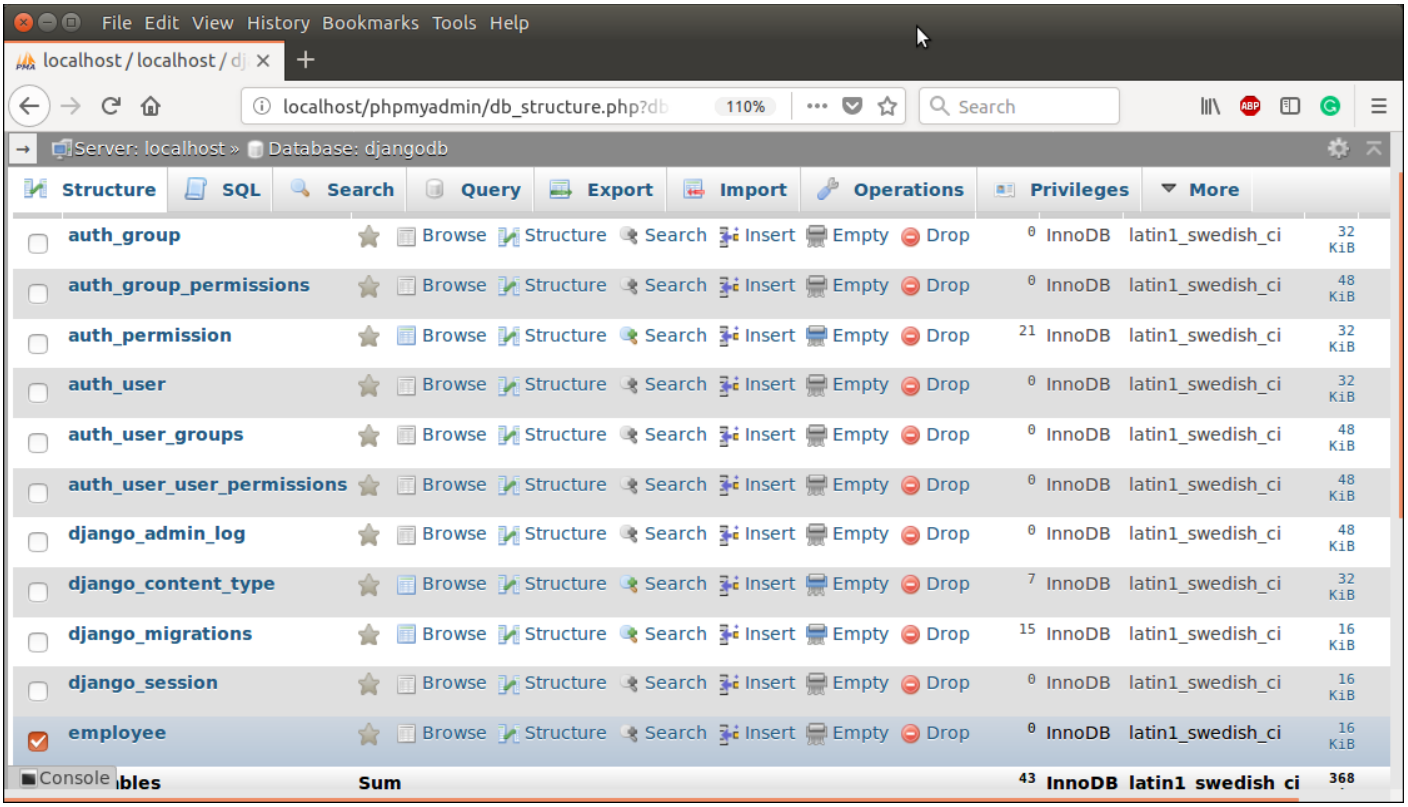
Run the command to migrate the migrations.

$ 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.

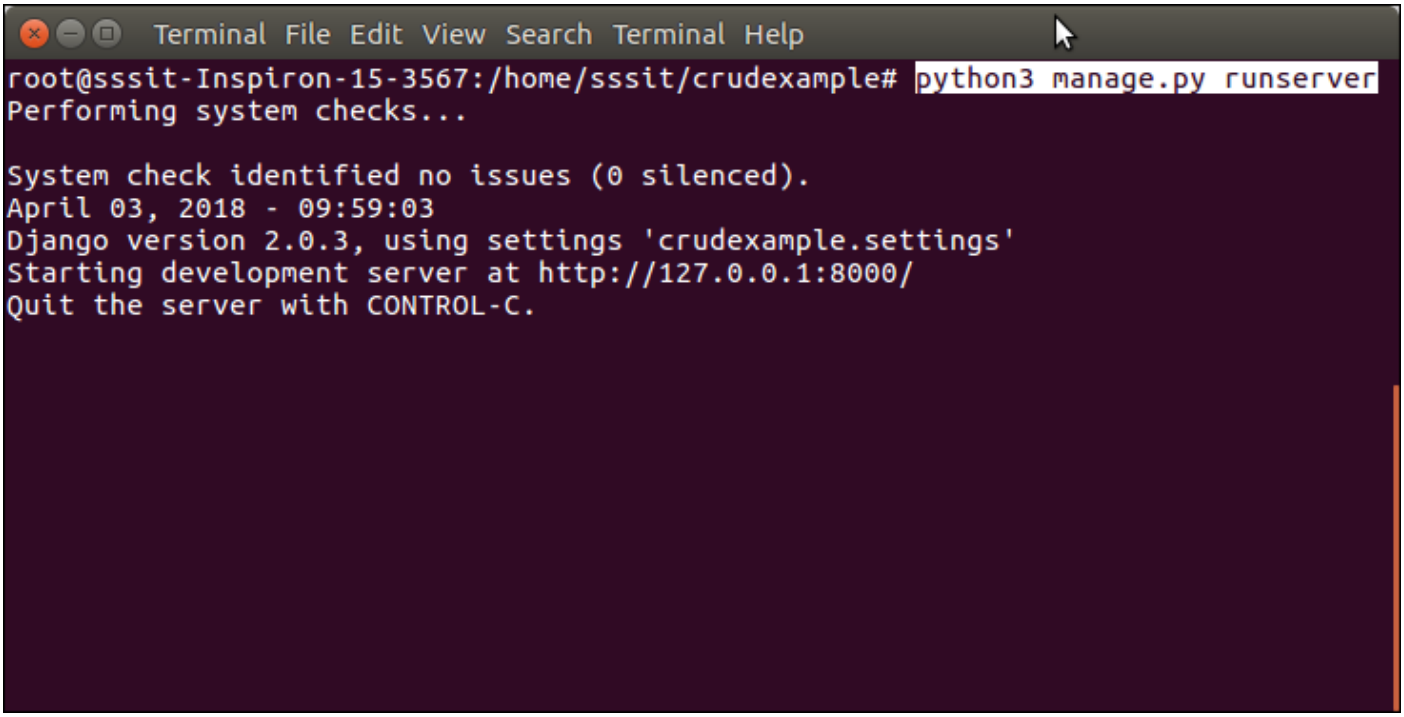
See list of tables created after migrate command.



Run Server

To run server use the following command.

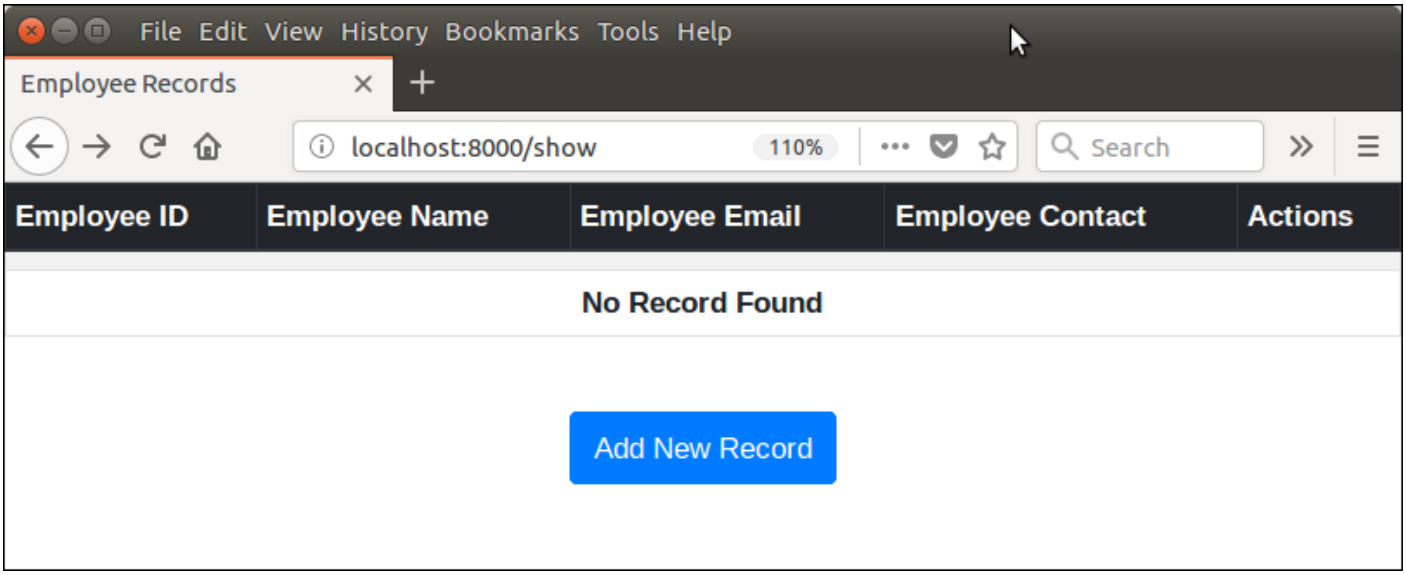
$ python3 manage.py runserver



Access to the Browser

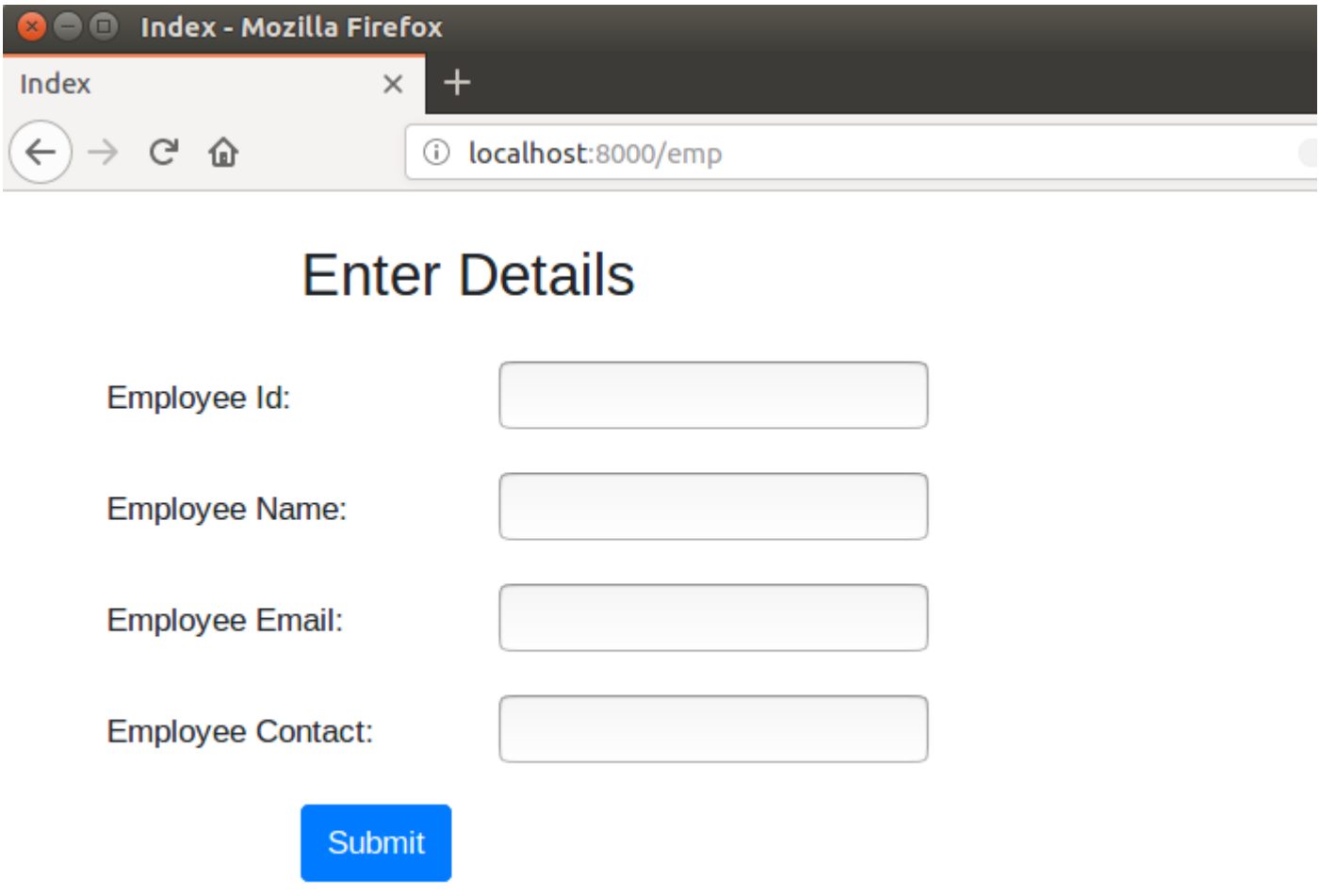
Access the application by entering **localhost:8000/show**, it will show all the available employee records.

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

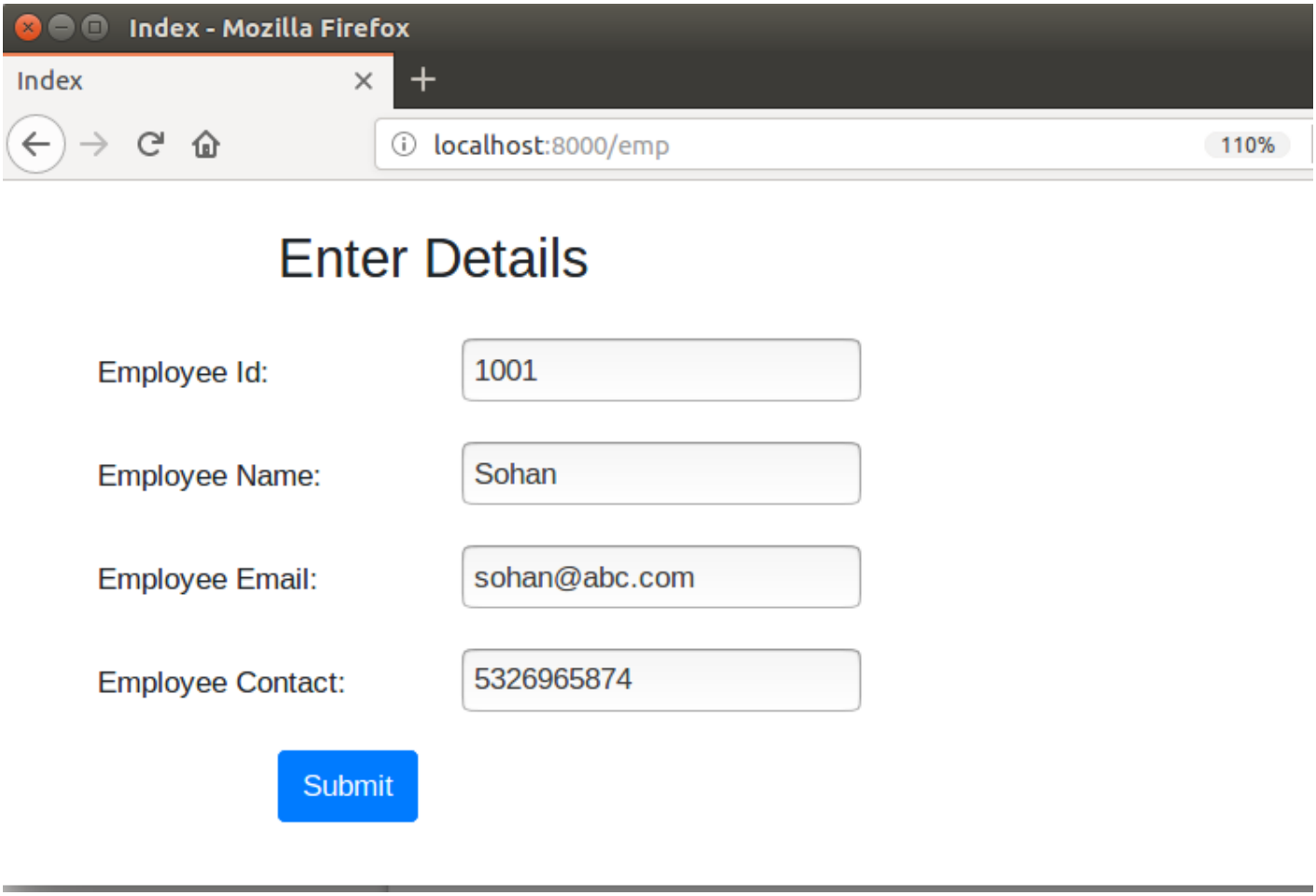


Adding Record

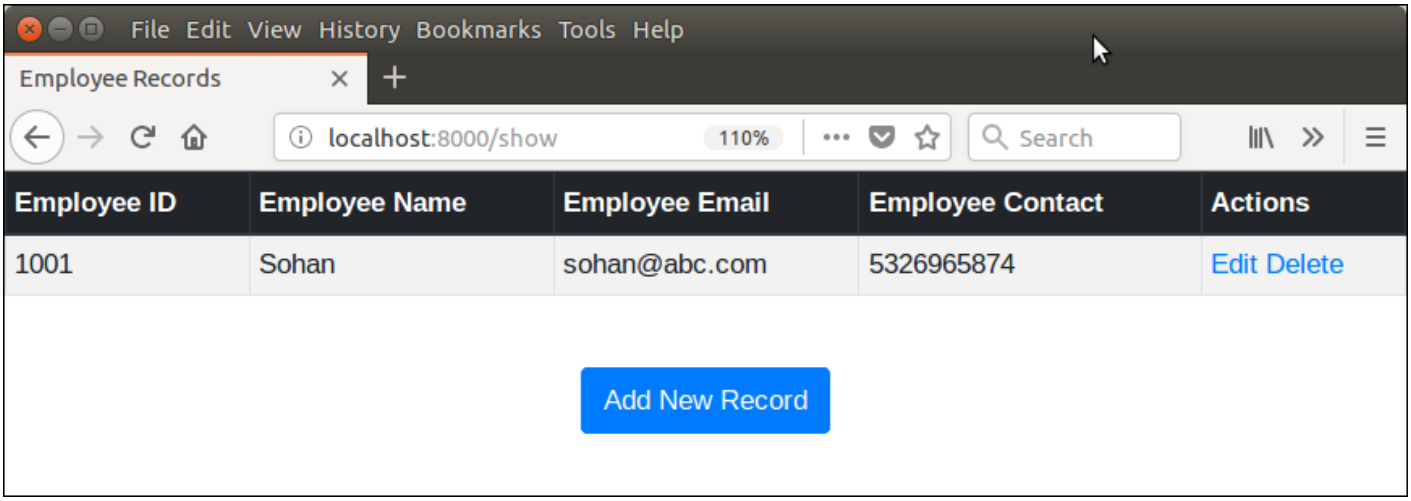
Click on the **Add New Record** button and fill the details. See the example.



Filling the details.

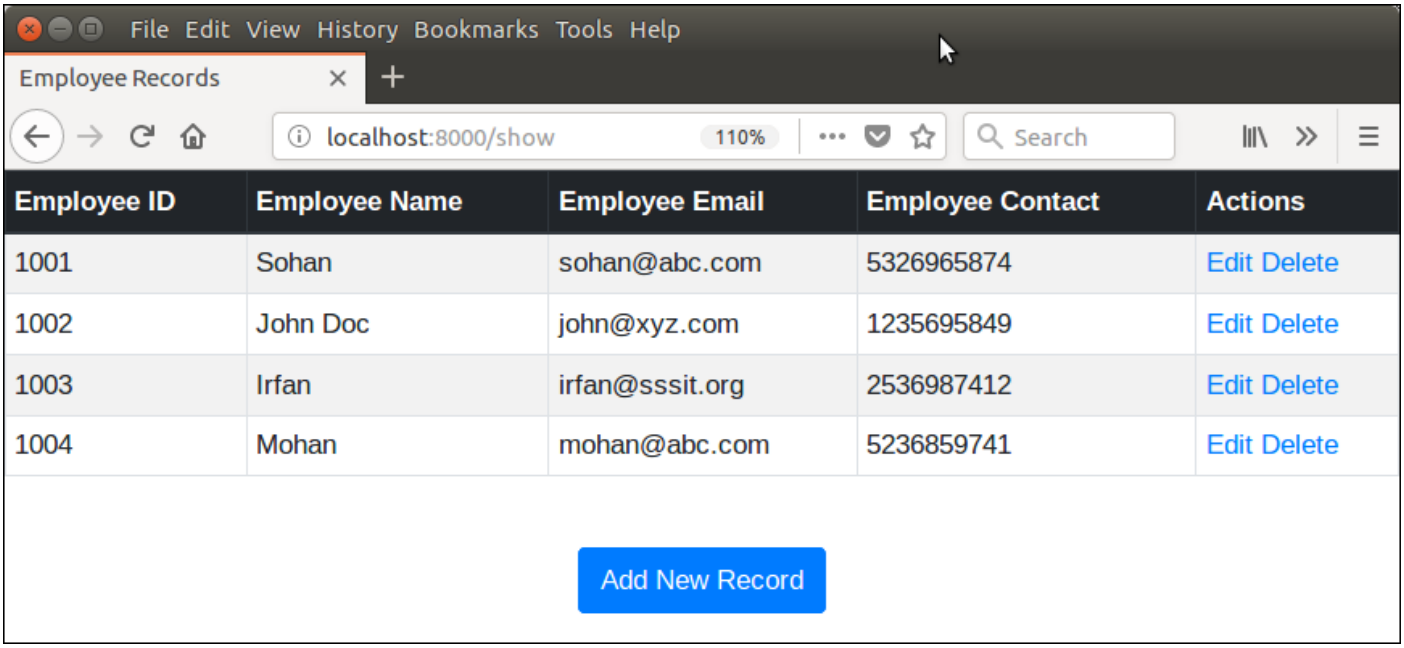


Submit the record and see, after submitting it shows the saved record.



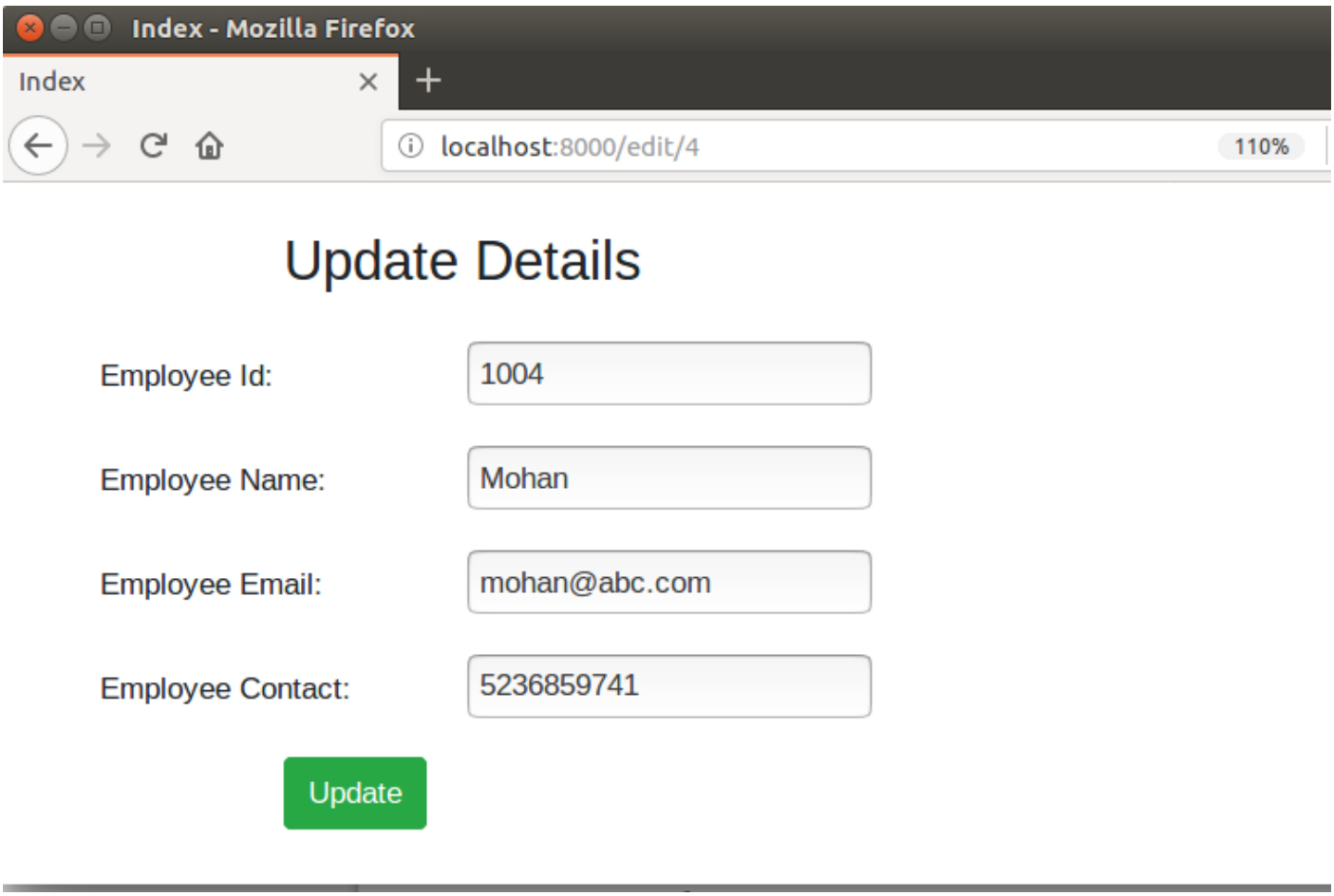
This section also allows, update and delete records from the **actions** column.

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

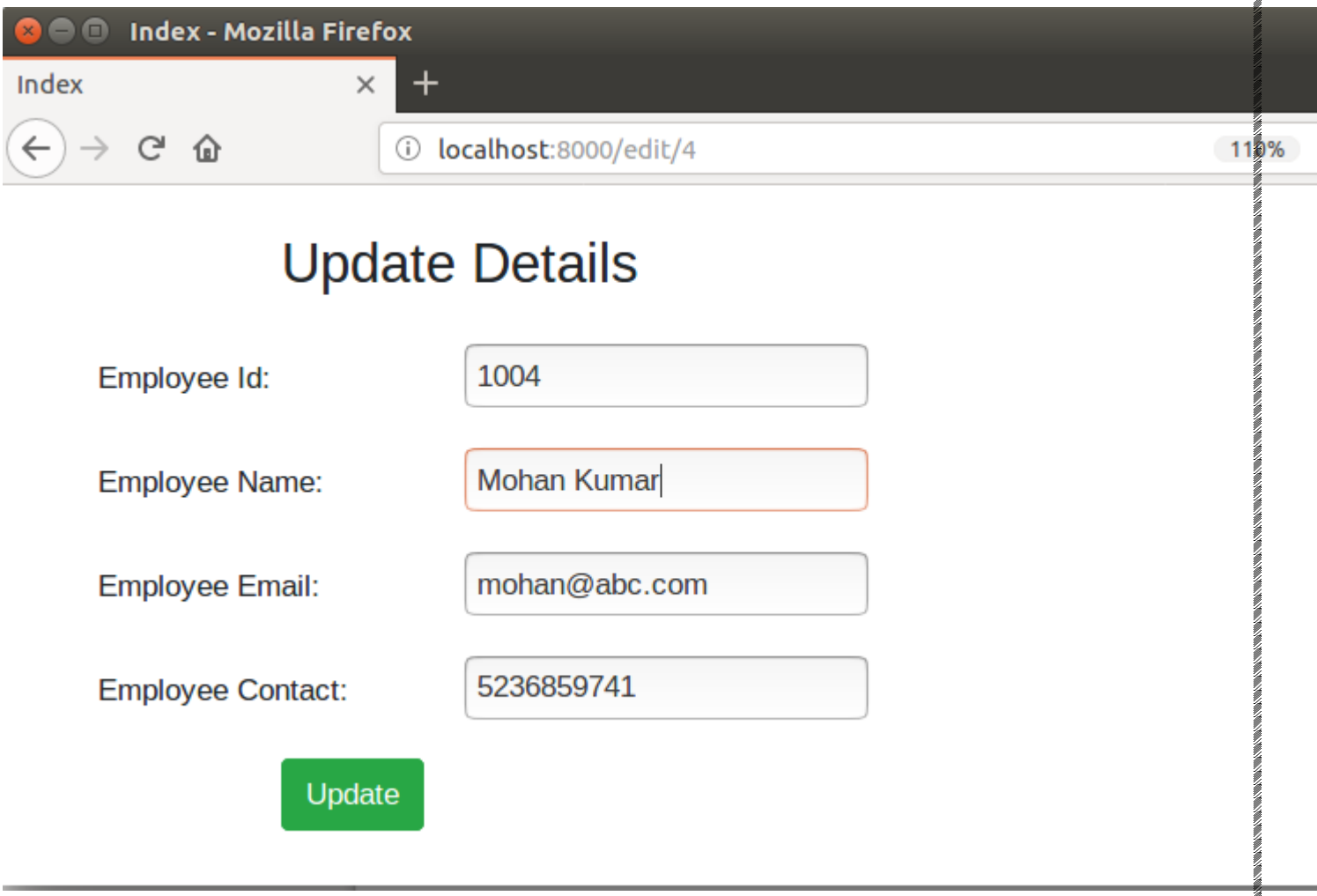


Update Record

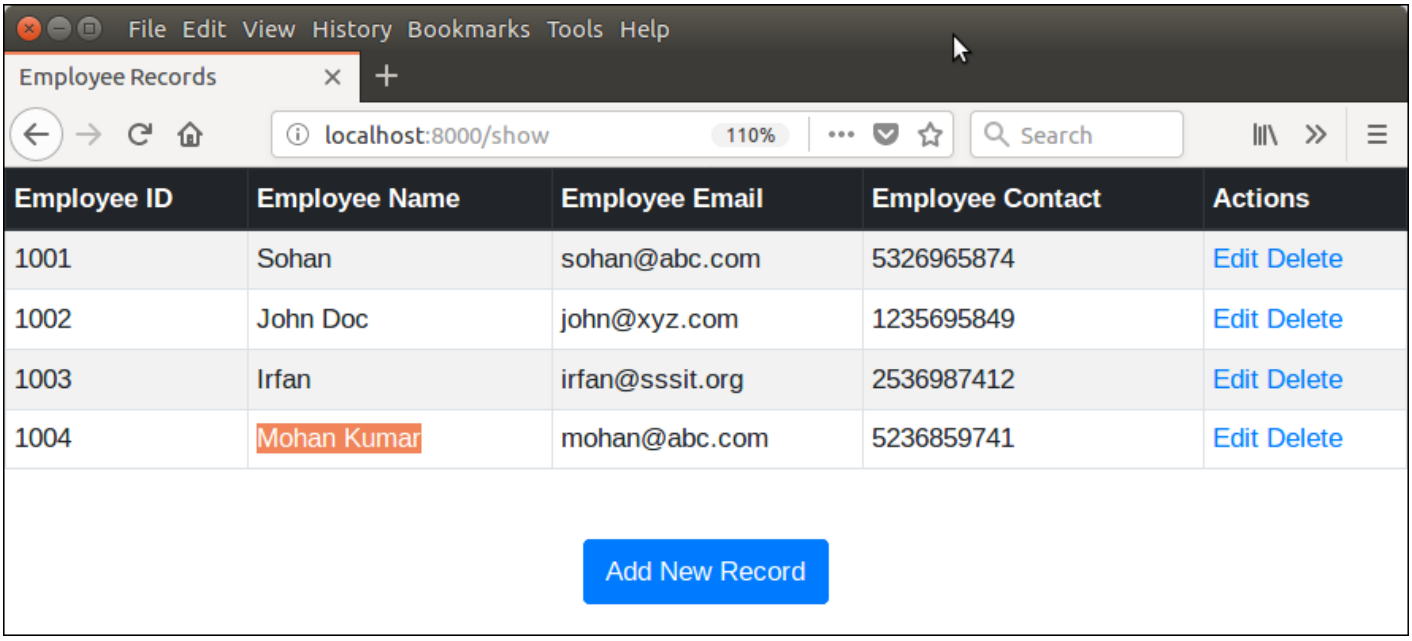
Lets update the record of **Mohan** by clicking on **edit** button. It will display record of Mohan in edit mode.



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



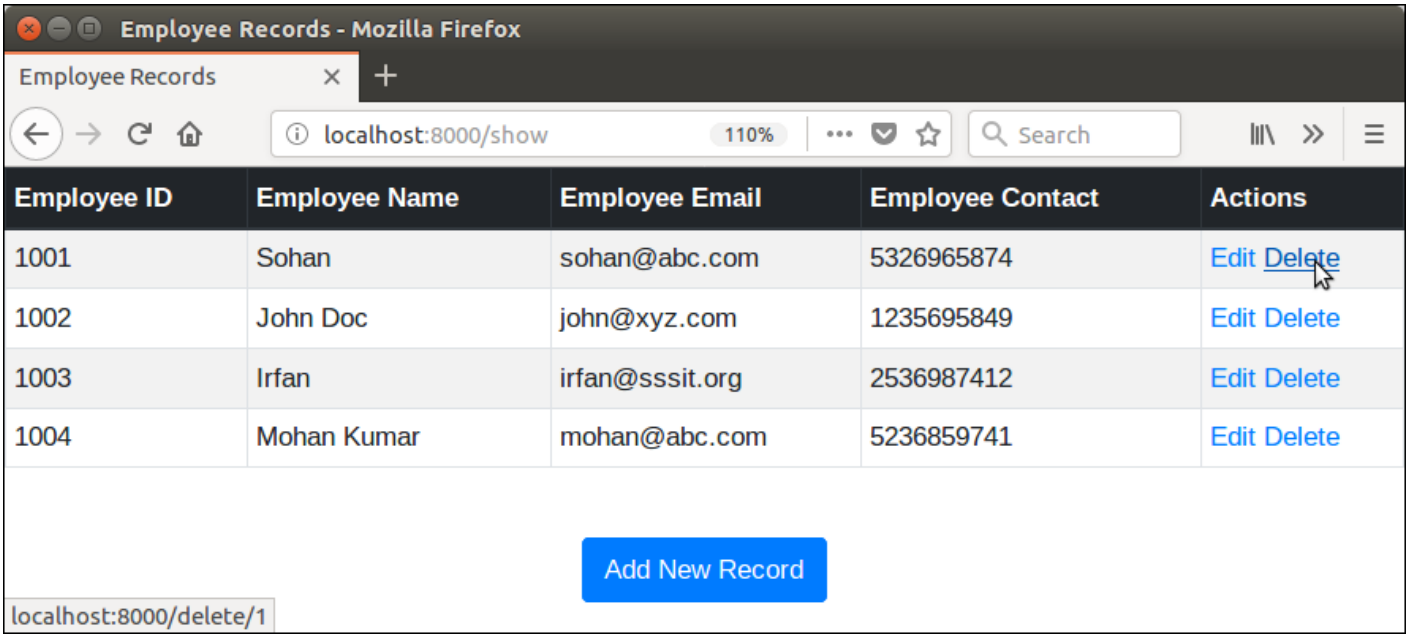
Click on update button and it redirects to the following page. See name is updated.



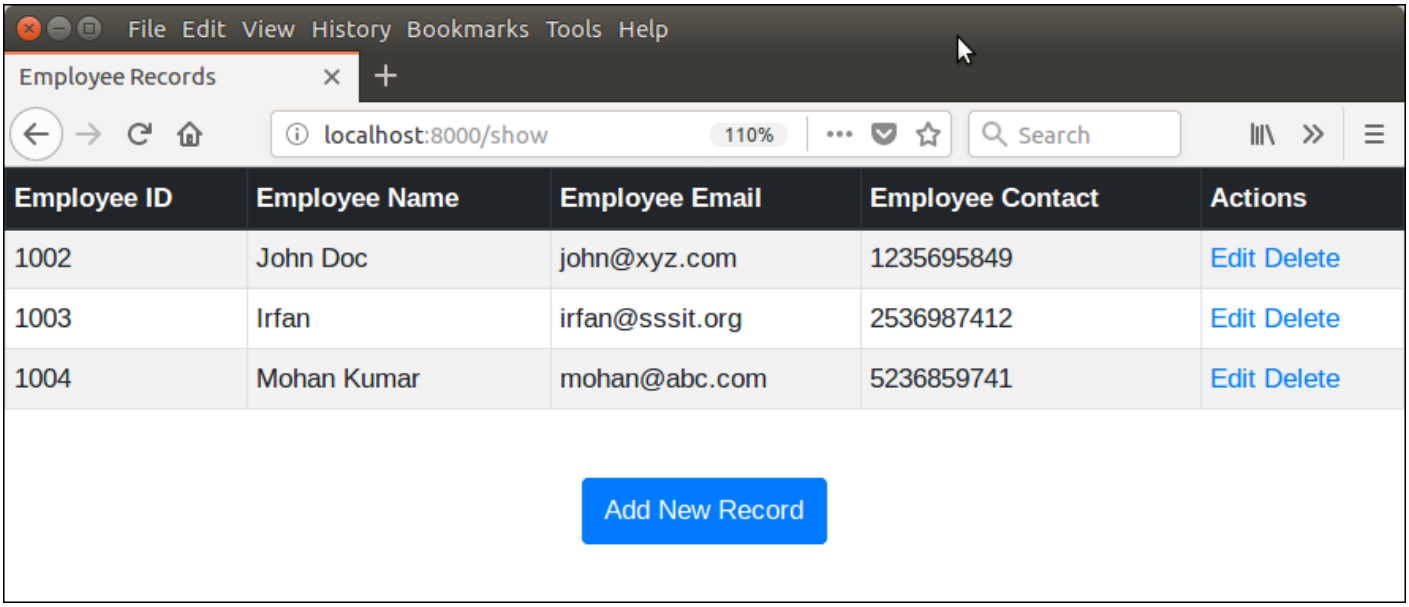
Same like, we can delete records too, by clicking the **delete** link.

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.