BLOG POST WEB APPLICATION

A Report for the Evaluation 3 of Project 2

Submitted by

AAKASH DEVRANI

(1613101007)

in partial fulfilment for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING

Under the Supervision of

Mr. MANOJ KUMAR, Assistant Professor

APRIL / MAY- 2020

SCHOOL OF COMPUTING AND SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Certified that this project report **"BLOG POST WEB APPLICATION"** is the bonafide work of **"AAKASH DEVRANI (1613101007)"** who carried the project work under my supervision.

SIGNATURE OF HEAD	SIGNATURE OF SUPERVISOR
Dr. MUNISH SHABARWAL	Mr. MANOJ KUMAR
PhD(Management),PhD(CS)	Assistant professor
Professor & dean	School of Computing Science
School of Computing Science	and Engineering
and Engineering	

ABSTRACT

The purpose of Online Blogging System is to automate the existing manual system by the help of computerized and full-fledged computer software, fulfilling their requirements, so that their valuable data/information can be stored for a longer period with easy accessing and manipulation of the same. The required software and hardware are easily available and easy to work with.

Online Blogging System, as described above, can lead to error free, secure, reliable and fast management system. It can assist the user to concentrate on their other activities rather to concentrate on the record keeping. Thus, it will help organization in better utilization of resources. The organization can maintain computerized records without redundant entries. That means that one need not be distracted by information that is not relevant, while being able to reach the information.

The aim is to automate its existing manual system by the help of computerized equipment and full-fledged computer software, fulfilling their requirements, so that their valuable data/information can be stored for a longer period with easy accessing and manipulation of the same. Basically, the project describes how to manage for good performance and better services for the clients.

It is also use for creating posting and viewing the others post, ideals, entries and blogs.

TABLE OF CONTENT

CHAPTER NO.	TITLES	PAGE NO.
	ABSTRACT	iii
	TABLE OF CONTENT	iv
	LIST OF FIGURES	v
01	INTRODUCTION	1
	1.1 Overall description	1
	1.2 Purpose	2
	1.2.1 Functionalities	2-3
	1.3 Motivations and scope	3
	1.3.1 Our project aims at Business process	3
	1.3.2 Requirements of project	4
02	PROBLEM STATEMENTS	4-5
03	EXISTING SYSTEM	5
	3.1 Drawback of Existing System	5
	3.2 Benefits of Project	6
04	PROPOSED MODEL	6
	4.1 Primary Design Phase	6
	4.2 Secondary Design Phase	7
05	IMPLEMENTATION/OUTPUT SCREENSHOTS	7
	5.1 User Interface Design	7
	5.1.1 Guidelines for User Interface Design	8-9
	5.1.2 Setup of virtual environment	9-12

	5.1.3 Database model	**12-14**
	5.1.4 Creating an Administrator Page	**14-15**
	5.1.5 Creating Post Model	**15-19**
	5.1.6 Adding URL pattens for Views (Post)	**19-20**
	5.1.7 Creating templates for Views	**20-22**
	5.1.8 Creating templates for Blog System	**22-24**
	5.1.9 Creating Sidebar	**25**
	5.1.10 Home Page	**26-28**
	5.1.11 Post Page	**28-29**
	5.1.12 Building Comment Model	**29-32**
	5.1.13 Building views (comment)	**32-34**
	5.1.14 Adding URL pattern for Views (comment)	**35-37**
	5.1.15 Making Comment Form Cripsy	**38**
06	**Conclusion/Future Enhancement**	**39-40**
07	**References**	**41-42**

TABLE OF FIGURES

S. No.	Fig. No.	Figure Contains	Page No.
01	5.1.2.1	Python Server	12
02	5.1.3	Database Models	14
03	5.1.4.1	Login Page	15
04	5.1.4.2	Admin Panel	15
05	5.1.5.1	Post Model	16
06	5.1.5.2	Blog Creation	17
07	5.1.5.3	Blog Post History	18
08	5.1.10.1	Homepage of Blog Application	28
09	5.1.11.1	Blog Posted by Random User	29
10	5.1.13.1	Comment Model View	34
11	5.1.13.2	Comment Page View	34

INTRODUCTION

The "Online Blogging System" has been developed to override the problems prevailing in the practicing manual system. This software is supported to eliminate, and in some cases reduce the hardships faced by this existing system. Moreover, this system is designed for the particular need for the company to carry out operations in a smooth and effective manner. This application is reduced as much as possible to avoid errors while entering the data. It also provides error message while entering invalid data. No formal knowledge is needed for the user to use this system. Thus, by this all it proves it is user-friendly. By using this can lead to error free, secure, reliable and fast management system.

1.1 Overall Description:

The main aim of this application is to provide a hassle-free accessing of the posted blogs, entries, topics etc. It also used for posting the blogs, editing the blogs, deleting the posted blogs etc. It is also used for viewing and posting the others one's blogs/posts.

Every organization, whether big or small, has challenges to overcome and managing the information of Idea, Blogs, Entries, Content, Views. Every Online Blogging System has different Blogs needs therefore we design exclusive employee management systems that are adapted to your managerial requirements. This is designed to assist in strategic planning, and will help you ensure that your organization is equipped with will allow you to manage your workforce anytime, at all times. These systems will ultimately allow you to better manage resources.

1.2 Purpose:

The main purpose of the Project on Online Blogging System is to manage the details of Blogs, Idea, Topic, Entries, Views. It manages all the information about Blogs, Content, Views, Blogs. The project is totally built at administrative end and thus only the administrator is guaranteed the access. The purpose of the Blogs, Idea, Content, Topic. It tracks all the details about the Topics, Entries, Views.

1.2.1 Functionalities provided by Online Blogging System are as follows:

- Provides the post viewing facility for everyone.
- Online Blogging System also manage the Content details online for Entries details, Views details, Blogs.
- It tracks all the information of Idea, Content, Entries etc.
- Manage the information of Idea.
- Shows the information and description of the Blogs, Topics.
- To increase efficiency of managing the Blogs, Topics.
- Manage the information of the Blogs.
- Editing, adding, posting and updating of Records.

1.3 Motivation and Scope:

The gist of the idea is to digitalize the process of manually driven ideas, topics and blog sharing among the user, small or big organizations as well as small forums And then to bake it into an web application to make it available for common people even to speed up process reducing further hassle and issues.

It may help collecting perfect management in details. In a very short time, the collection will be obvious, simple and sensible. It will help a person to know the management of passed year perfectly and vividly. It also helps in current all works relative to Online Blogging System. It will be also reduced the cost of collecting the management & collection procedure will go on smoothly.

1.3.1 Our project aims at Business process automation, i.e. we have tried to computerize various processes of Online Blogging System.

- To utilize resources in an efficient manner by increasing their productivity through automation.
- It satisfies the user requirement.
- Be easy to understand by the user and operator
- Be easy to operate.
- Have a good user interface
- Be expandable

- Delivered on schedule within the budget.

1.3.2 The proposed system has the following requirements:

- System needs store information about new entry of Blogs.
- System needs to help the internal staff to keep information of Idea and find them as per various queries.
- System needs to maintain quantity record.
- System need to update and delete the record.
- System also needs a search area.
- It also needs a security system to prevent data.

02. PROBLEM STATEMENT:

The old manual system was suffering from a series of drawbacks. Since whole of the system was to be maintained with hands the process of keeping, maintaining and retrieving the information was very tedious and lengthy. The records were never used to be in a systematic order. There used to be lots of difficulties in associating any particular transaction with a particular context. There would always be unnecessary consumption of time while entering records and retrieving records. One more problem was that it was very difficult to find errors while entering the records. Once the records were entered it was very difficult to update these records.

The reason behind it is that there is lot of information to be maintained and have to be kept in mind while running the business. For this reasons we have provided features Present

system is partially automated, actually existing system is quite laborious as one has to enter same information at three different places.

03. EXISTING SYSTEM:

Existing system is manual system. It requires lots of file work to be done. It is a time consuming system. All user information is maintained manually. Any searching requires so much effort manually. There is no way of spreading the information so fast and in cheapest manner in previous system all information does not get into one place, here people can write whatever they want to write.

3.1 DRAWBACKS OF EXISTING SYSTEM:

1) Data redundancy and formatting – the various files are likely to have different formats and therefore lead to redundancy and inconsistency.
2) Maintaining registers is costly – traditionally documents have been stored in batches and they filed in file cabinets and boxes. A numerical system is assigned specifically a user number assigned to organize the files.
3) Error prone – existing system are error prone, since manual work is required. More time is consumed and errors may propagate due to human mistakes.
4) Low security feature – due to maintenance of record manually and shared and could view easily by anyone also these could be possible loss of data and confidential information due to some disaster in form of fire, theft etc.

3.2 BENEFITS OF PROJECT:

- Data can be saved safely
- No other person can view other persons detail
- Greater efficiency
- User friendliness
- Minimum time required
- Free of cost

04. PROPOSED MODEL:

In this phase, a logical system is built which fulfills the given requirements. Design phase of software development deals with transforming the client's requirements into a logically working system. Normally, design is performed in the following in the following two steps:

4.1 Primary Design Phase:

In this phase, the system is designed at block level. The blocks are created on the basis of analysis done in the problem identification phase. Different blocks are created for different functions emphasis is put on minimizing the information flow between blocks. Thus, all activities which require more interaction are kept in one block.

4.2 Secondary Design Phase:

In the secondary phase the detailed design of every block is performed.

4.3 The general tasks involved in the design process are the following:

- Design various blocks for overall system processes.
- Design smaller, compact and workable modules in each block.
- Design various database structures.
- Specify details of programs to achieve desired functionality.
- Design the form of inputs, and outputs of the system.
- Perform documentation of the design.
- System reviews.

05. IMPLEMENTATION:

5.1 User Interface Design:

User Interface Design is concerned with the dialogue between a user and the computer. It is concerned with everything from starting the system of logging into the system to the eventually presentation of desired inputs and outputs. The overall flow of screens and messages is called a dialogue.

5.1.1 The following steps are various guidelines for User Interface Design:

- The system user should always be aware of what to do next.

- The screen should be formatted so that various types of information, instructions and messages always appear in the same general display area.
- Message, instructions or information should be displayed long enough to allow the system user to read them.
- Use display attributes sparingly.
- Default values for fields and answered to be entered by the user should be specified.
- A use should not be allowed to proceed without correcting an error.
- The system user should never get an operating system message or fatal error.

5.1.2 *Setup of virtual environment:*

A virtual environment is a self-contained directory tree that contains dependencies required by different projects isolated to existing packages.

By using virtual Python environments, applications can run in their own 'sandbox' in isolation of other Python applications

Step 1:

Open your terminal and create a directory to store all your virtual environments, using the command "mkdir Environments" which is an acronym of "make directory".

Now go inside the directory using the command CD which stands for call Directory, "CD Environments"

Step 2:

Now we will use a module named virtualenv to create isolated virtual environments.

But before it first, install this module

"pip install virtualenv"

If error occurs then install pip package manager first.

To verify a successful installation run this

"virtualenv –version"

- *To create a Virtual Environment for Python 3*

  python3 -m venv myenv

  Or

  virtualenv myenv

  Or

  $ "*python -m venv myenv"*

  This will create a directory myenv along with directories inside it containing a copy of the Python interpreter, the standard library, and various supporting files.

  A virtual Python environment has a similar directory structure to a global Python installation. The bin directory contains executables for the virtual environment, the include directory is linked to the global Python installation header files, the lib directory is a copy of the global Python installation libraries and where packages for the virtual environment are installed, and the shared directory is used to place shared Python packages.

- *To add modules and packages in our Environment, we need to activate it first.*

  On Windows, run: "myenv\Script\activate.bat"

  On Unix or MacOs, run: "source myenv/bin/activate"

- *To deactivate the current Environment, run:*

$ “*deactivate*”

Code snippet:

Now run the following command in your shell to create a Django project.

$ “*django-admin startproject mysite*”

This will generate a project structure with several directories and python scripts.

```
├── mysite
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── wsgi.py
├── manage.py
```

To navigate the python server manager, run:

$ “*cd mysite*”
$ “*python manage.py startapp blog*”
These will create an app named blog in our project.

```
├── db.sqlite3
├── mysite
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── wsgi.py
├── manage.py
└── blog
    ├── __init__.py
```

```
├── admin.py
├── apps.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

Now we need to inform Django that a new application has been created, open your `settings.py` file and scroll to the installed apps section, which should have some already installed apps.

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

Now add the newly created app blog at the bottom and save it.

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

Next, make migrations.

$ "*python manage.py migrate*"
This will apply all the unapplied migrations on the SQLite database which comes along with the Django installation.

Let's test our configurations by running the Django's built-in development server.

$ "*python manage.py runserver*"
Open your browser and go to this address `http://127.0.0.1:8000/` if everything went well you should see this page.

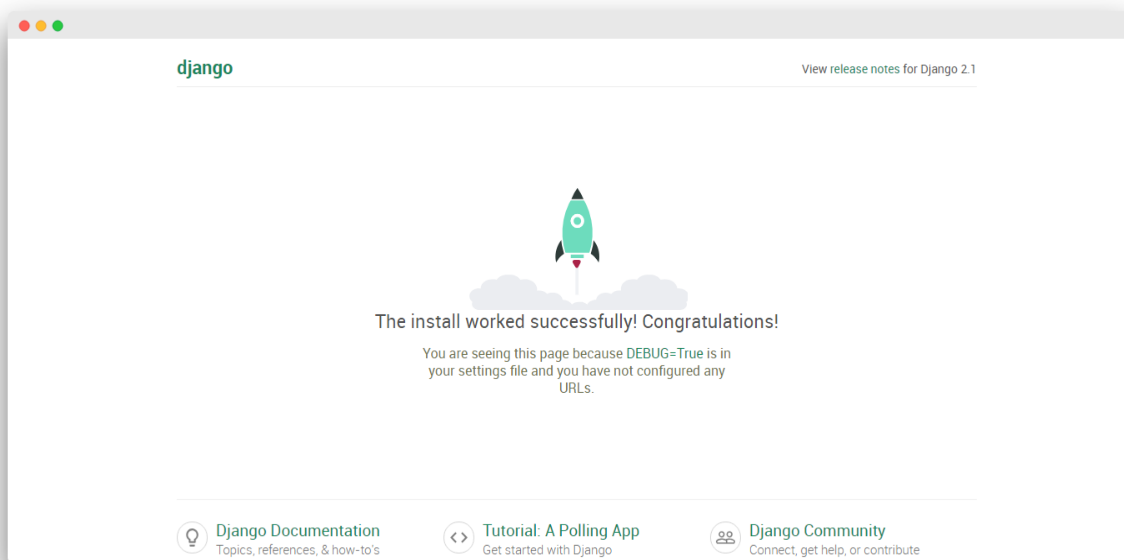

Fig. 5.1.2.1

5.1.3 Database Models

Now we will define the data models for our blog. A model is a Python class that subclasses `django.db.models.Model`, in which each attribute represents a database field. Using this subclass functionality, we automatically have access to everything within django.db.models.Models and can add additional fields and methods as desired. We will have a Post model in our database to store posts.

```
from django.db import models

from django.contrib.auth.models import User
```

```
STATUS = (
    (0,"Draft"),
    (1,"Publish")
)

class Post(models.Model):
    title = models.CharField(max_length=200, unique=True)
    slug = models.SlugField(max_length=200, unique=True)
    author = models.ForeignKey(User, on_delete=
models.CASCADE,related_name='blog_posts')
    updated_on = models.DateTimeField(auto_now= True)
    content = models.TextField()
    created_on = models.DateTimeField(auto_now_add=True)
    status = models.IntegerField(choices=STATUS, default=0)

    class Meta:
        ordering = ['-created_on']

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

At the top, we're importing the class `models` and then creating a subclass of `models.Model` Like any typical blog, each blog post will have a title, slug, author name, and the timestamp or date when the article was published or last updated.
Notice how we declared a tuple for `STATUS` of a post to keep draft and published posts separated when we render them out with templates.
The Meta class inside the model contains metadata. We tell Django to sort results in the `created_on` field in descending order by default when we query the database. We specify descending order using the negative prefix. By doing so, posts published recently will appear first.
The `__str__()` method is the default human-readable representation of the object. Django will use it in many places, such as the administration site.
Now that our new database model is created we need to create a new migration record for it and migrate the change into our database.

(django) $ python manage.py makemigrations
(django) $ python manage.py migrate
Now we are done with the database.

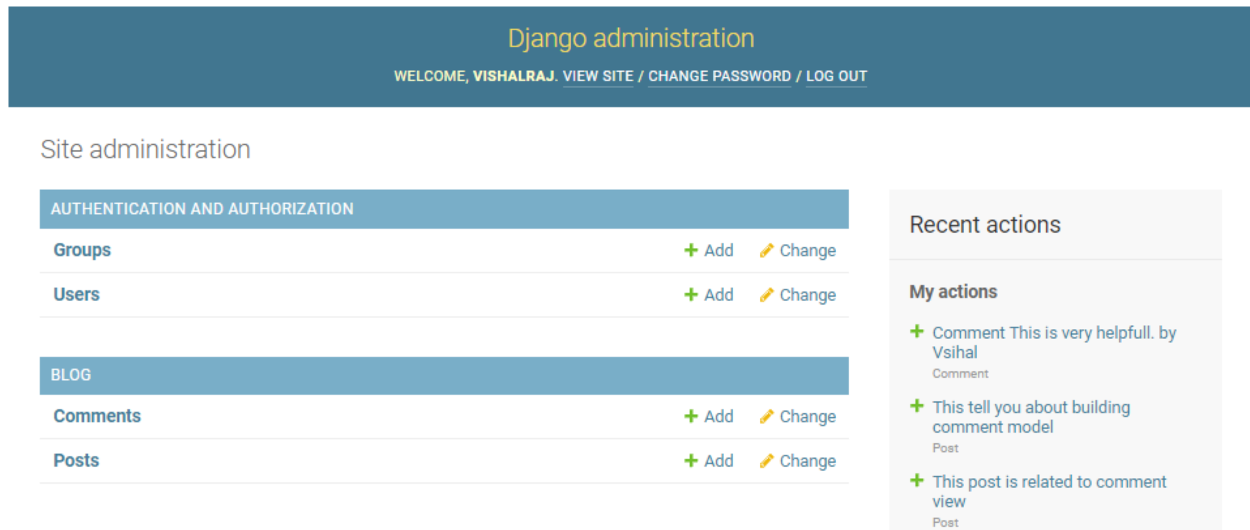

Fig. 5.1.3.1

5.1.4 Creating an Administrator Page:

We will create an admin panel to create and manage Posts. Fortunately, Django comes with an inbuilt admin interface for such tasks.

In order to use the Django admin first, we need to create a superuser by running the following command in the prompt.

$ "python manage.py createsuperuser"
You will be prompted to enter email, password, and username. Note that for security concerns Password won't be visible.

Username (leave blank to use 'user'): admin

Email address: admin@gamil.com

Password:
Password (again):
Enter any details you can always change them later. After that rerun the development server and go to the address `http://127.0.0.1:8000/admin/`
$ "*python manage.py runserver"*
You should see a login page, enter the details you provided for the superuser.

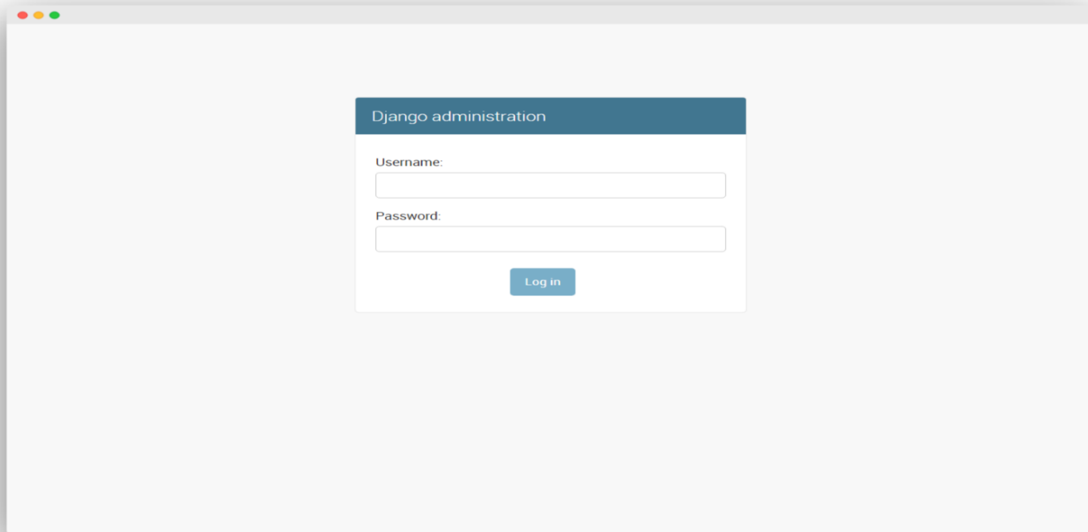

Fig. 5.1.4.1

After you log in you should see a basic admin panel with Groups and Users models which come from Django authentication framework located in `django.contrib.auth`.

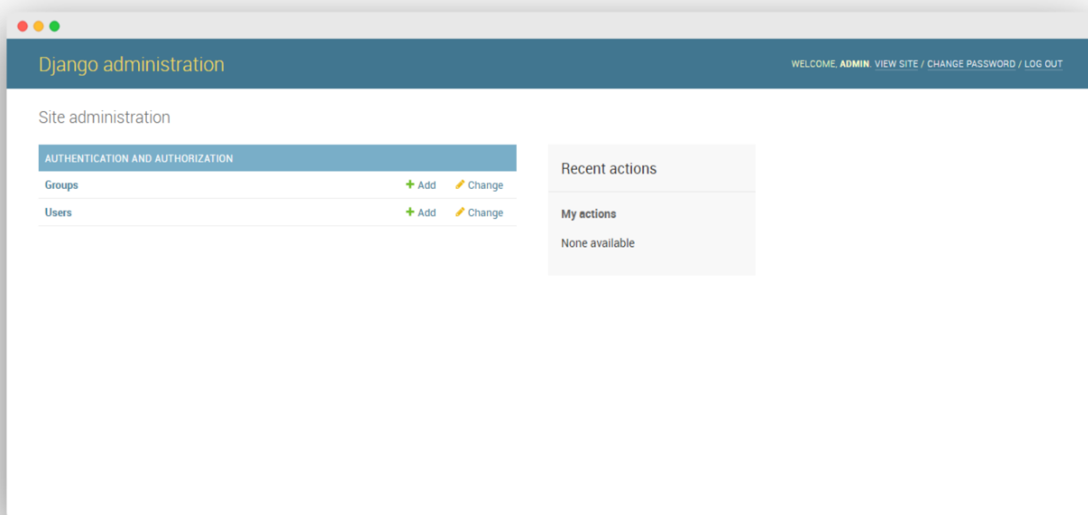

Fig. 5.1.4.2

Still, we can't create posts from the panel we need to add the Post model to our admin.

5.1.5 Post Model:

Adding Models To The Administration Site

Open the `blog/admin.py` file and register the Post model there as follows.

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Save the file and refresh the page you should see the Posts model there.

Now let's create our first blog post click on the Add icon beside Post which will take you to another page where you can create a post. Fill the respective forms and create your first ever post.

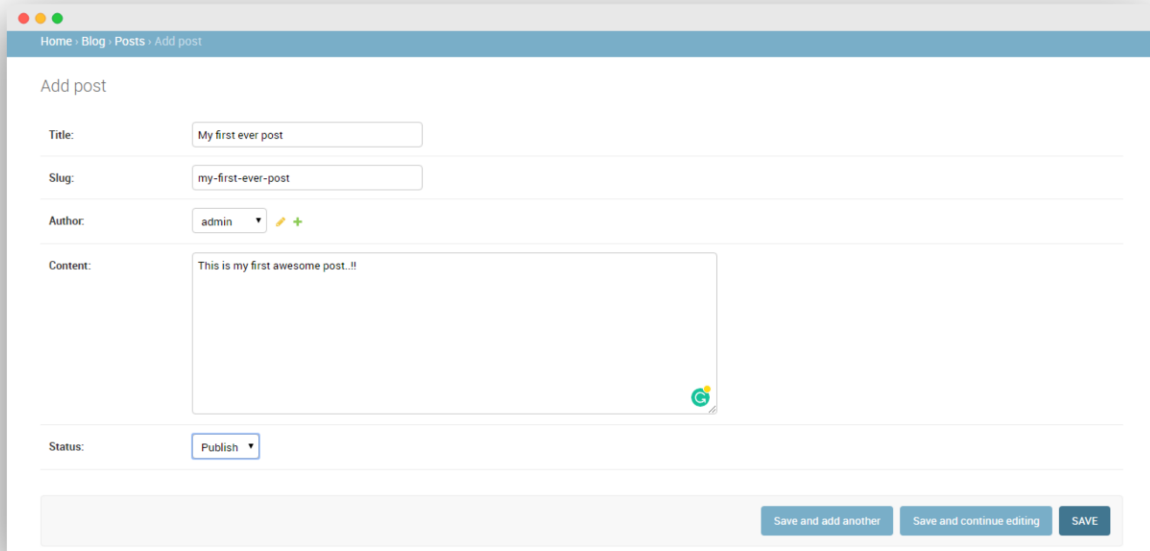

Fig. 5.1.5.1

Once you are done with the Post save it now, you will be redirected to the post list page with a success message at the top.

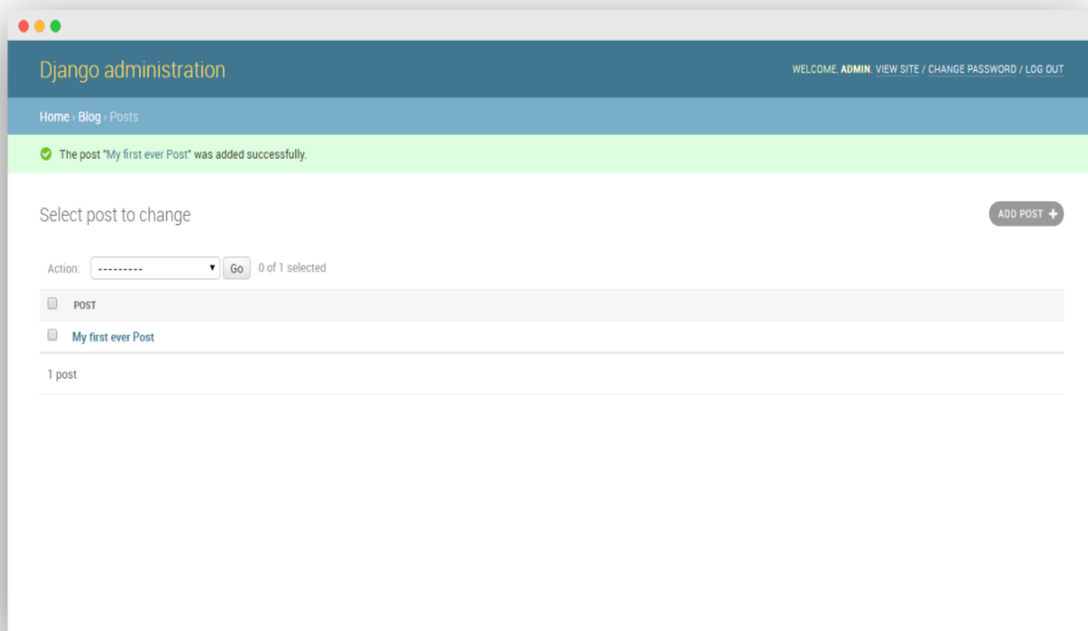

Fig. 5.1.5.2

Though it does the work, we can customize the way data is displayed in the administration panel according to our convenience. Open the `admin.py` file again and replace it with the code below.

```
from django.contrib import admin
from .models import Post

class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'status','created_on')
    list_filter = ("status",)
    search_fields = ['title', 'content']
    prepopulated_fields = {'slug': ('title',)}

admin.site.register(Post, PostAdmin)
```

This will make our admin dashboard more efficient. Now if you visit the post list, you will see more details about the Post.

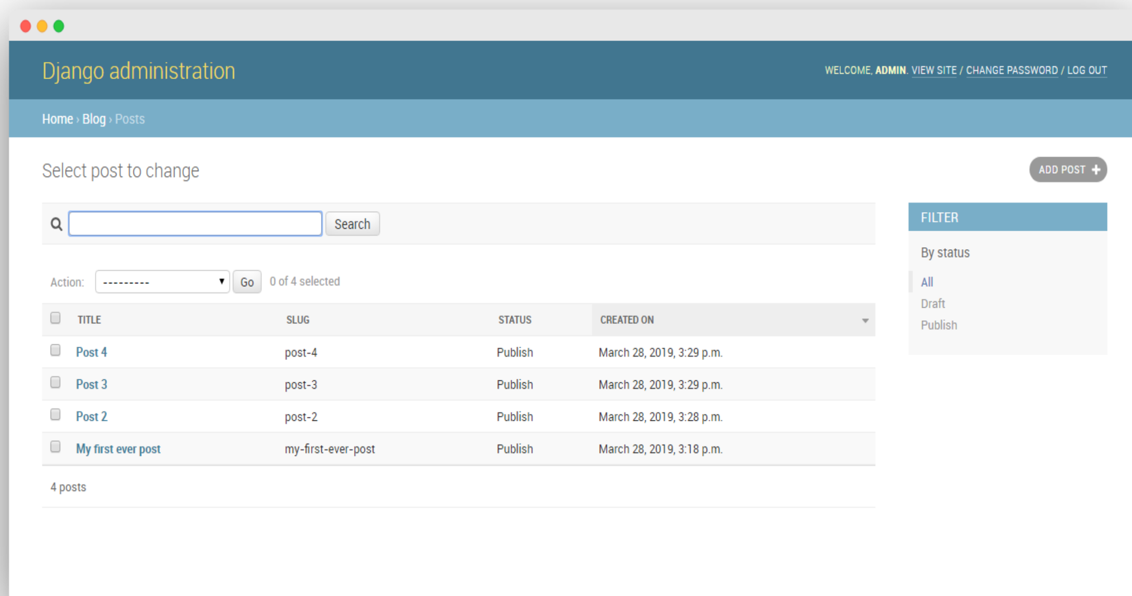

Fig. 5.1.5.3

Note that I have added a few posts for testing.

The `list_display` attribute does what its name suggests display the properties mentioned in the tuple in the post list for each post.
If you notice at the right, there is a filter which is filtering the post depending on their Status this is done by the `list_filter` method.
And now we have a search bar at the top of the list, which will search the database from the `search_fields` attributes. The last attribute `prepopulated_fields` populates the slug, now if you create a post the slug will automatically be filled based upon your title.
Now that our database model is complete we need to create the necessary views, URLs, and templates so we can display the information on our web application.

Building Views

A Django view is just a Python function that receives a web request and returns a web response.
We're going to use class-based views then map URLs for each view and create an HTML templated for the data returned from the views.
Open the `blog/views.py` file and start coding.

```
from django.views import generic
```

```
from .models import Post


class PostList(generic.ListView):
    queryset = Post.objects.filter(status=1).order_by('-created_on')
    template_name = 'index.html'


class PostDetail(generic.DetailView):
    model = Post
    template_name = 'post_detail.html'
```

The built-in ListViews which is a subclass of generic class-based-views render a list with the objects of the specified model we just need to mention the template, similarly DetailView provides a detailed view for a given object of the model at the provided template.

Note that for `PostList` view we have applied a filter so that only the post with status published be shown at the front end of our blog. Also in the same query, we have arranged all the posts by their creation date. The (–) sign before the `created_on` signifies the latest post would be at the top and so on.

5.1.6 Adding URL patterns for Views

We need to map the URL for the views we made above. When a user makes a request for a page on your web app, the Django controller takes over to look for the corresponding view via the `urls.py` file, and then return the HTML response or a 404 not found error, if not found. Create an `urls.py` file in your blog application directory and add the following code.

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


urlpatterns = [
    path('', views.PostList.as_view(), name='home'),
    path('<slug:slug>/', views.PostDetail.as_view(), name='post_detail'),
]
```

We mapped general URL patterns for our views using the path function. The first pattern takes an empty string denoted by `' '` and returns the result generated from the `PostList` view which

is essentially a list of posts for our homepage and at last we have an optional parameter name which is basically a name for the view which will later be used in the templates.
Names are an optional parameter, but it is a good practice to give unique and rememberable names to views which makes our work easy while designing templates and it helps keep things organized as your number of URLs grows.

Next, we have the generalized expression for the `PostDetail` views which resolve the slug (a string consisting of ASCII letters or numbers) Django uses angle brackets `< >` to capture the values from the URL and return the equivalent post detail page.
Now we need to include these blog URLs to the actual project for doing so open the `mysite/urls.py` file.

```
from django.contrib import admin


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

Now first import the include function and then add the path to the new `urls.py` file in the URL patterns list.

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


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

Now all the request will directly be handled by the blog app.

5.1.7 Creating Templates For The Views

We are done with the Models and Views now we need to make templates to render the result to our users. To use Django templates we need to configure the template setting first.
Create directory templates in the base directory. Now open the project's `settings.py` file and just below `BASE_DIR` add the route to the template directory as follows.

```
TEMPLATES_DIR = os.path.join(BASE_DIR,'templates')
```

Now In `settings.py` scroll to the,`TEMPLATES` which should look like this.

```
TEMPLATES = [
```

```
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        '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',
            ],
        },
    },
]
```

Now add the newly created `TEMPLATE_DIRS` in the `DIRS`.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        #  Add  'TEMPLATE_DIRS' here
        'DIRS': [TEMPLATE_DIRS],
        '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',
            ],
        },
    },
]
```

Now save and close the file we are done with the configurations.

Django makes it possible to separate python and HTML, the python goes in views and HTML goes in templates. Django has a powerful template language that allows you to specify how data is displayed. It is based on template tags, template variables, and template filters.

5.1.8 Creating templates for our Blog Web System

I'll start off with a `base.html` file and a `index.html` file that inherits from it. Then later when we add templates for homepage and post detail pages, they too can inherit from `base.html`. Let's start with the base.html file which will have common elements for the blog at any page like the navbar and footer. Also, we are using Bootstrap for the UI and Roboto font.

File base.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Django Central</title>
    <link href="https://fonts.googleapis.com/css?family=Roboto:400,700" rel="stylesheet">
    <meta name="google" content="notranslate" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
      crossorigin="anonymous" />
  </head>

  <body>
    <style>
      body {
      font-family: "Roboto", sans-serif;
```

```
        font-size: 17px;
        background-color: #fdfdfd;
      }
      .shadow {
        box-shadow: 0 4px 2px -2px rgba(0, 0, 0, 0.1);
      }
      .btn-danger {
        color: #fff;
        background-color: #f00000;
        border-color: #dc281e;
      }
      .masthead {
        background: #3398E1;
        height: auto;
        padding-bottom: 15px;
        box-shadow: 0 16px 48px #E3E7EB;
        padding-top: 10px;
      }
    </style>

      <!-- Navigation -->
      <nav class="navbar navbar-expand-lg navbar-light bg-light shadow" id="mainNav">
        <div class="container-fluid">
          <a class="navbar-brand" href="{% url 'home' %}">Django central</a>
          <button class="navbar-toggler navbar-toggler-right" type="button" data-
toggle="collapse" data-target="#navbarResponsive"
            aria-controls="navbarResponsive" aria-expanded="false" aria-label="Toggle
navigation">
            <span class="navbar-toggler-icon"></span>
          </button>
          <div class="collapse navbar-collapse" id="navbarResponsive">
```

```
                <ul class="navbar-nav ml-auto">
                    <li class="nav-item text-black">
                        <a class="nav-link text-black font-weight-bold" href="#">About</a>
                    </li>
                    <li class="nav-item text-black">
                        <a class="nav-link text-black font-weight-bold" href="#">Policy</a>
                    </li>
                    <li class="nav-item text-black">
                        <a class="nav-link text-black font-weight-bold" href="#">Contact</a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>
    {% block content %}
    <!-- Content Goes here -->
    {% endblock content %}
    <!-- Footer -->
    <footer class="py-3 bg-grey">
        <p class="m-0 text-dark text-center ">Copyright &copy; Django Central</p>
    </footer>
</body>
</html>
```

This is a regular HTML file except for the tags inside curly braces `{ }` these are called template tags.
The `{% url 'home' %}` Returns an absolute path reference, it generates a link to the home view which is also the List view for posts.
The `{% block content %}` Defines a block that can be overridden by child templates, this is where the content from the other HTML file will get injected.

5.1.9 Creating a Sidebar for all the pages

Next, we will make a small sidebar widget which will be inherited by all the pages across the site. Notice sidebar is also being injected in the `base.html` file this makes it globally available for pages inheriting the base file.

```
{% block sidebar %}
<style>
        .card{
           box-shadow: 0 16px 48px #E3E7EB;
        }
</style>

<!-- Sidebar Widgets Column -->
<div class="col-md-4 float-right ">
<div class="card my-4">
        <h5 class="card-header">About Us</h5>
    <div class="card-body">
        <p class="card-text"> This awesome blog is made on the top of our Favourite full stack
Framework 'Django', follow up the tutorial to learn how we made it..!</p>
        <a href="https://djangocentral.com/building-a-blog-application-with-django"
          class="btn btn-danger">Know more!</a>
    </div>
</div>
</div>
{% endblock sidebar %}
```

5.1.10 Creating the Home Page

Next, create the `index.html` file of our blog that’s the homepage.

```
{% extends "base.html" %}
{% block content %}
<style>
   body {
       font-family: "Roboto", sans-serif;
       font-size: 18px;
       background-color: #fdfdfd;
   }

   .head_text {
       color: white;
   }

   .card {
       box-shadow: 0 16px 48px #E3E7EB;
   }
</style>

<header class="masthead">
   <div class="overlay"></div>
   <div class="container">
       <div class="row">
           <div class=" col-md-8 col-md-10 mx-auto">
               <div class="site-heading">
                   <h3 class=" site-heading my-4 mt-3 text-white"> Welcome to my awesome Blog
</h3>
                   <p class="text-light">We Love Django As much as you do..!  
                   </p>
               </div>
           </div>
       </div>
```

```
    </div>
</header>
<div class="container">
    <div class="row">
        <!-- Blog Entries Column -->
        <div class="col-md-8 mt-3 left">
            {% for post in post_list %}
            <div class="card mb-4">
                <div class="card-body">
                    <h2 class="card-title">{{ post.title }}</h2>
                    <p class="card-text text-muted h6">{{ post.author }} | {{ post.created_on}} </p>
                    <p class="card-text">{{post.content|slice:":200" }}</p>
                    <a href="{% url 'post_detail' post.slug  %}" class="btn btn-primary">Read More
&rarr;</a>
                </div>
            </div>
            {% endfor %}
        </div>
        {% block sidebar %} {% include 'sidebar.html' %} {% endblock sidebar %}
    </div>
</div>
{%endblock%}
```

With the `{% extends %}` template tag, we tell Django to inherit from the base.html template. Then, we are filling the content blocks of the base template with content.
Notice we are using for loop in HTML that's the power of Django templates it makes HTML Dynamic. The loop is iterating through the posts and displaying their title, date, author, and body, including a link in the title to the canonical URL of the post.
In the body of the post, we are also using template filters to limit the words on the excerpts to 200 characters. Template filters allow you to modify variables for display and look like `{{ variable | filter }}`.
Now run the server and visit `http://127.0.0.1:8000/` you will see the homepage of our blog.

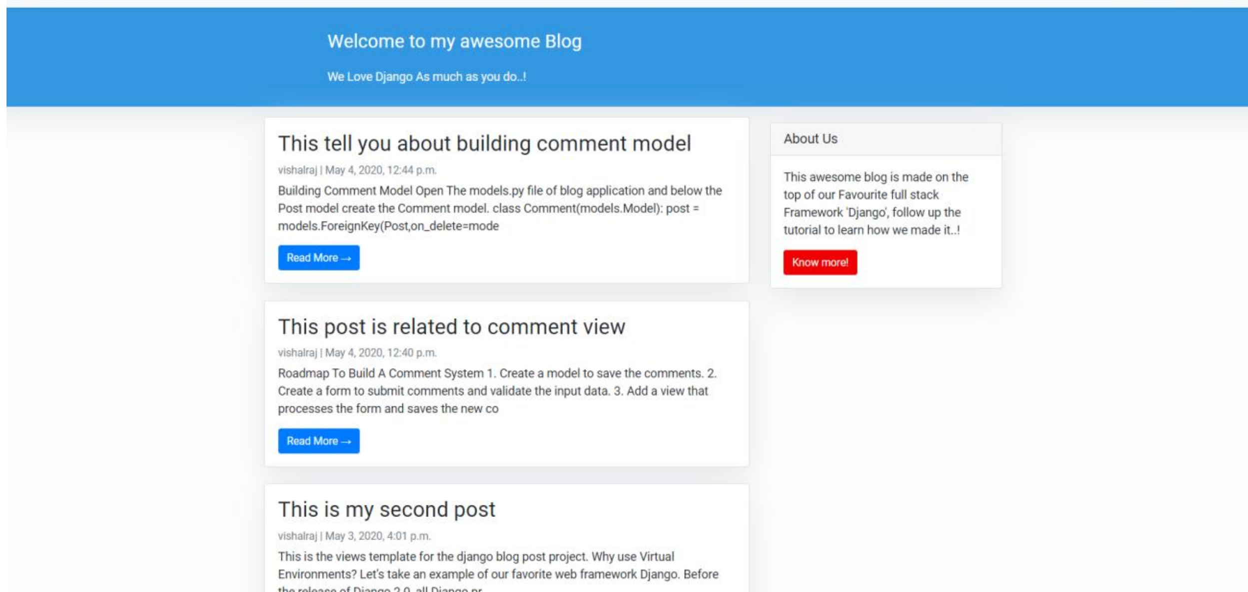

Fig. 5.1.10.1

Looks good..!

5.1.11 Creating a Post Page

Now let’s make an HTML template for the detailed view of our posts.

Next, Create a file `post_detail.html` and paste the below HTML there.

```
{% extends 'base.html' %} {% block content %}
<div class="container">
  <div class="row">
    <div class="col-md-8 card mb-4  mt-3 left  top">
      <div class="card-body">
        <h1>{% block title %} {{ object.title }} {% endblock title %}</h1>
        <p class=" text-muted">{{ post.author }} | {{ post.created_on }}</p>
        <p class="card-text ">{{ object.content | safe }}</p>
      </div>
```

```
    </div>
    {% block sidebar %} {% include 'sidebar.html' %} {% endblock sidebar %}
  </div>
</div>
{% endblock content %}
```

At the top, we specify that this template inherits from.`base.html` Then display the `body` from our context object, which `DetailView` makes accessible as an object.
Now visit the homepage and click on read more, it should redirect you to the post detail page.

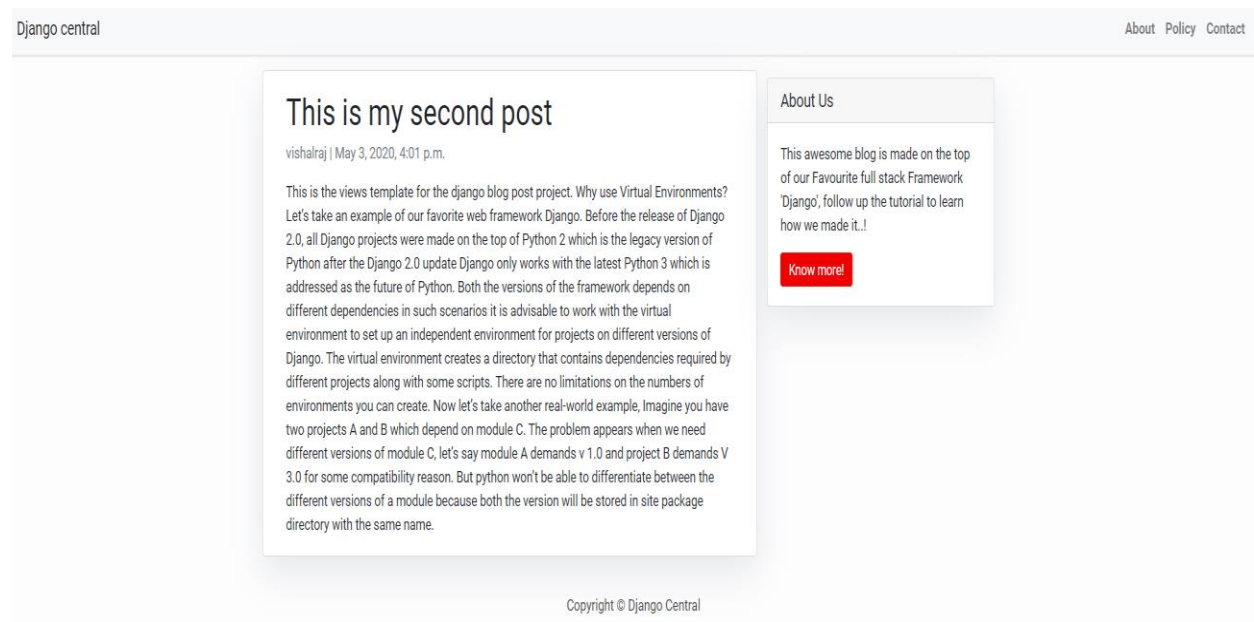

Fig. 5.1.11.1

5.1.12 Building Comment Model

Open The models.py file of blog application and below the Post model create the Comment model.

```
class Comment(models.Model):
    post = models.ForeignKey(Post,on_delete=models.CASCADE,related_name='comments')
    name = models.CharField(max_length=80)
```

```
    email = models.EmailField()
    body = models.TextField()
    created_on = models.DateTimeField(auto_now_add=True)
    active = models.BooleanField(default=False)

    class Meta:
        ordering = ['created_on']

    def __str__(self):
        return 'Comment {} by {}'.format(self.body, self.name)
```

In this comment model first, we have a Foreign key relation that establishes a many-to-one relationship with the Post model, since every comment will be made on a post and each post will have multiple comments.
The related_name attribute allows us to name the attribute that we use for the relation from the related object back to this one. After defining this, we can retrieve the post of a comment object using comment.post and retrieve all comments of a post using post.comments.all() . If you don't define the related_name attribute, Django will use the name of the model in lowercase, followed by _set (that is, comment_set) to name the manager of the related object back to this one.
As a traditional comment system, we are accepting the commenter's name, email and comment body as inputs. Then we have an active boolean field that is set to False to prevent spam we will manually allow all the comments posted.
The Meta class inside the model contains metadata. We tell Django to sort results in the created_on field in descending order by default when we query the database. We specify descending order using the negative prefix. By doing so, comments made recently will appear first.
The __str__() method is the default human-readable representation of the object. Django will use it in many places, such as the administration site.
Next, we need to synchronize this comment model into the database by running migrations to reflects the changes in the database.

```
(django) $ python manage.py makemigrations
(django) $ python manage.py migrate
```

Adding Comments Model To The Administration Site

Open `admins.py` file and write the following code.

```
from django.contrib import admin
from .models import Post, Comment
@admin.register(Comment)
class CommentAdmin(admin.ModelAdmin):
    list_display = ('name', 'body', 'post', 'created_on', 'active')
    list_filter = ('active', 'created_on')
    search_fields = ('name', 'email', 'body')
    actions = ['approve_comments']


    def approve_comments(self, request, queryset):
        queryset.update(active=True)
```

Going over the code `@admin.register(Comment)` registers the comment into the Admin area. Below the `CommentAdmin` class to customizes the representation of data on the screen.
The `list_display` attribute does what its name suggests display the properties mentioned in the tuple in the comments list for each comment.
The `list_filter` method will filter the comments based on the creation date and their active status and `search_fields` will simply search the database for the parameters provided in the tuple.
Finally, we have the actions method this will help us for approving many comment objects at once, the `approve_comments` method is a simple function that takes a queryset and updates the active boolean field to `True`.
Now create a superuser if you haven't already and log in to the dashboard you should see the comment model there.
Now click on comments and create your comments.

Creating forms from models

Django offers a very rich and secure API to handle forms. Since the form input will be saved in the database models we are gonna use the Django's **ModelForm**.
A common practice is to create a `forms.py` file inside your app directory for all the forms of an app. So create a `forms.py` file in your app and write the following code.

```
from .models import Comment
```

```python
from django import forms


class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ('name', 'email', 'body')
```

In the model form, we just need to provide the model name in the `Meta` class of the form Django will handle the form processing and validation on the basis of fields of the model. By default, Django will generate a form dynamically from all fields of the model but we can explicitly define the fields we want the forms to have, that is what `fields` attribute is doing here.

5.1.13 Building Views

We will modify the post detail view for form processing using function based view.

```python
from .models import Post
from .forms import CommentForm
from django.shortcuts import render, get_object_or_404

def post_detail(request, slug):
    template_name = 'post_detail.html'
    post = get_object_or_404(Post, slug=slug)
    comments = post.comments.filter(active=True)
    new_comment = None
    # Comment posted
    if request.method == 'POST':
        comment_form = CommentForm(data=request.POST)
        if comment_form.is_valid():

            # Create Comment object but don't save to database yet
            new_comment = comment_form.save(commit=False)
```

```
            # Assign the current post to the comment
            new_comment.post = post
            # Save the comment to the database
            new_comment.save()
    else:
        comment_form = CommentForm()

    return render(request, template_name, {'post': post,
                                           'comments': comments,
                                           'new_comment': new_comment,
                                           'comment_form': comment_form})
```

This post detail view will show the post and all its comments, let's break it down to see what's happening.

First, we assigned the HTML template to a variable name `template_name` for future reference and we are assigning the Post object inside the `post` variable.
This `comments = post.comments.filter(active=True)` queryset retrieves all the approved comments from the database.
Since this is the same page where users will create new comments we initialized the `new_comment` variable by setting it to none.
Next, we have a conditional statement if a `POST` request is made, the comment_form variable will hold the data of user input next Django will validate the data using the `is_valid()` method.
If the form is valid the following actions take place.

1. We create a new Comment object by calling the form's `save()` method and assign it to the `new_comment` variable, but with `commit=Flase` which will prevent it from saving into the database right away because we still have to link it the post object
2. We assigned the comment object to the current post
3. Finally, save the object into the database

Else if it is a `GET` request we initialize the form object and pass it to the template.

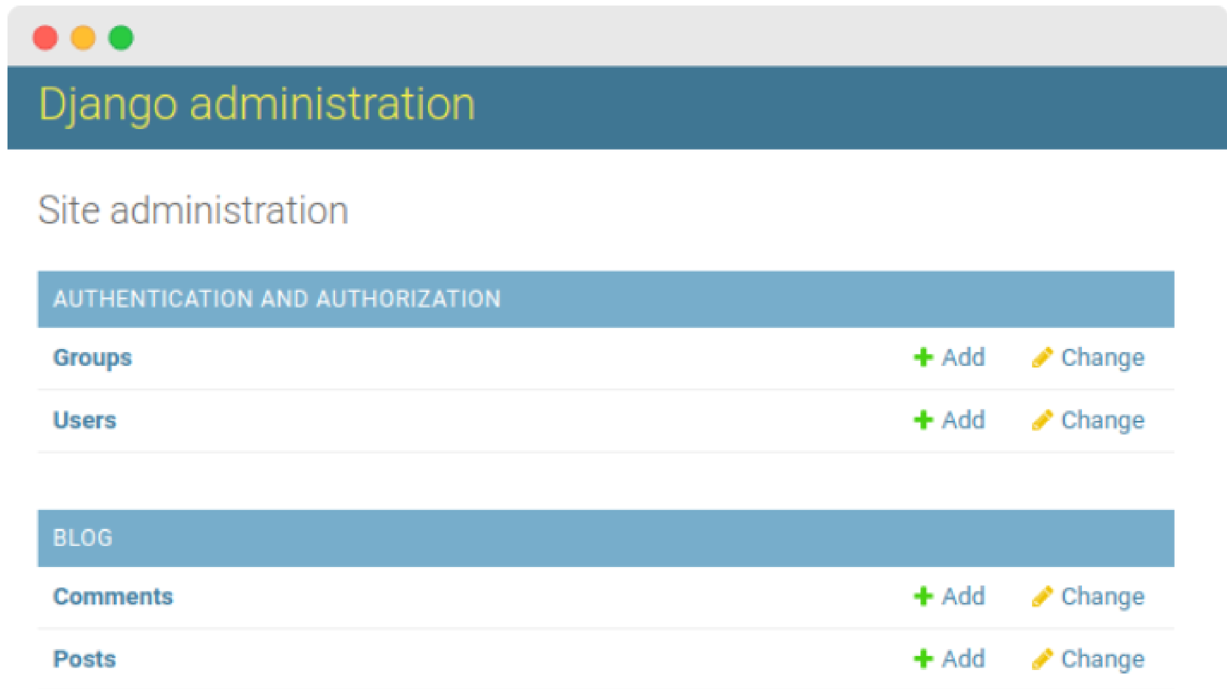

Fig. 5.1.13.1

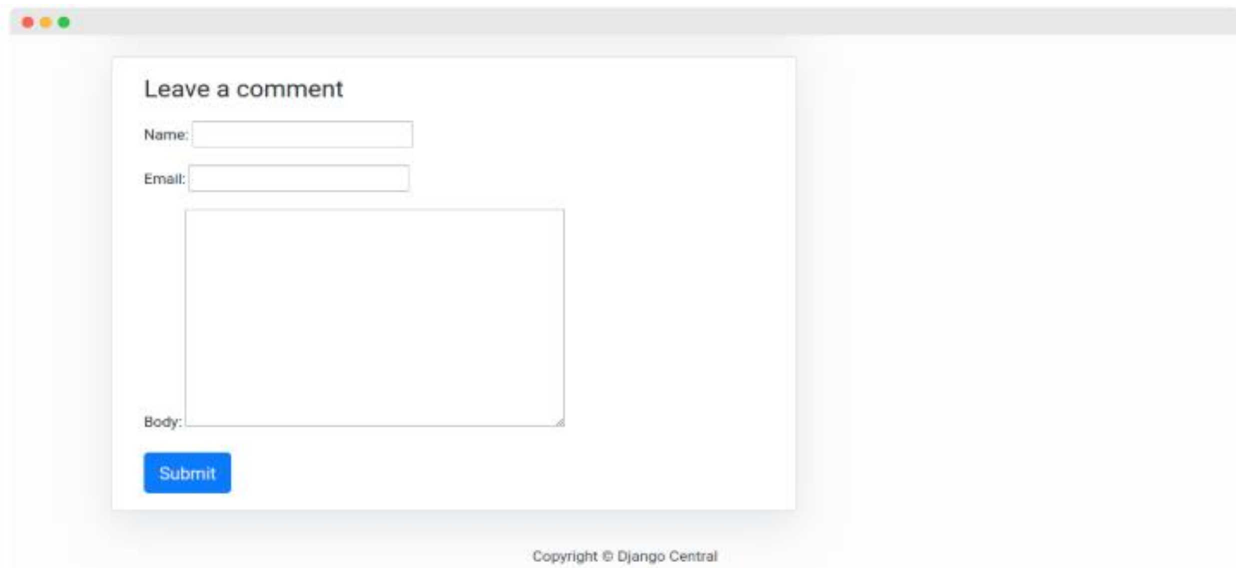

Fig. 5.1.13.2

5.1.14 Adding URL patterns for Views

Open the `urls.py` file of your app and map the view.
path('<slug:slug>/', views.post_detail, name='post_detail')

Creating Templates For The Views

Let's see what we will do in the templates.

```
  {% for comment in comments %}
      <div class="comments" style="padding: 10px;">
        <p class="font-weight-bold">
          {{ comment.name }}
          <span class=" text-muted font-weight-normal">
            {{ comment.created_on }}
          </span>
        </p>
        {{ comment.body | linebreaks }}
      </div>
      {% endfor %}
```

Here we are using Django's `{% for %}` template tag for looping over comments, then for each comment object, we are displaying the user's name,creation date and the comment body.

```
 <div class="card-body">
      {% if new_comment %}
      <div class="alert alert-success" role="alert">
        Your comment is awaiting moderation
      </div>
      {% else %}
      <h3>Leave a comment</h3>
```

```
    <form method="post" style="margin-top: 1.3em;">
      {{ comment_form.as_p }}
      {% csrf_token %}
      <button type="submit" class="btn btn-primary  btn-lg">Submit</button>
    </form>
    {% endif %}
  </div>
```

When a user makes a new comment we show them a message saying, “Your comment is awaiting moderation” else we render the form.

So putting the entire template all together we have this,

```
{% extends 'base.html' %} {% block content %}

<div class="container">
  <div class="row">
    <div class="col-md-8 card mb-4  mt-3 left  top">
      <div class="card-body">
        <h1>{% block title %} {{ post.title }} {% endblock title %}</h1>
        <p class=" text-muted">{{ post.author }} | {{ post.created_on }}</p>
        <p class="card-text ">{{ post.content | safe }}</p>
      </div>
    </div>
    {% block sidebar %} {% include 'sidebar.html' %} {% endblock sidebar %}

    <div class="col-md-8 card mb-4  mt-3 ">
      <div class="card-body">
        <!-- comments -->
        <h2>{{ comments.count }} comments</h2>

        {% for comment in comments %}
```

```
        <div class="comments" style="padding: 10px;">
          <p class="font-weight-bold">
            {{ comment.name }}
            <span class=" text-muted font-weight-normal">
              {{ comment.created_on }}
            </span>
          </p>
          {{ comment.body | linebreaks }}
        </div>
        {% endfor %}
      </div>
    </div>
    <div class="col-md-8 card mb-4  mt-3 ">
      <div class="card-body">
        {% if new_comment %}
        <div class="alert alert-success" role="alert">
          Your comment is awaiting moderation
        </div>
        {% else %}
        <h3>Leave a comment</h3>
        <form method="post" style="margin-top: 1.3em;">
          {{ comment_form.as_p }}
          {% csrf_token %}
          <button type="submit" class="btn btn-primary  btn-lg">Submit</button>
        </form>
        {% endif %}
      </div>
    </div>
  </div>
</div>
{% endblock content %}
```

5.1.15 Making Comment Form Crispy

Although our form works as expected, yet we can make the form look better without much changing the template using the Django crispy form library. It's a very popular library for form managing you can check it out here
Install it using

```
pip install django-crispy-forms
```

Add it to the installed apps list.

```
INSTALLED_APPS = [
    ...


    'crispy_forms',
]
```

For Bootstrap 4 styling add this in settings.py file.

```
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

Now in your template, you just need to load the crispy tag and use the crispy tag beside the form, as follows.

```
{% load crispy_forms_tags %}
...
<form method="post" style="margin-top: 1.3em;">
        {{ comment_form | crispy }}
        {% csrf_token %}
        <button type="submit" class="btn btn-primary  btn-lg">Submit</button>
 </form>
```

6. CONCLUSION:

While developing a system a conscious effort has been made to create and develop a software package, making use of available tools, techniques and resources – that will generate the proper system cases.

While making the system an eye has been kept on making it as user friendly as such one may hope that the system will be acceptable to any user and will adequetly meet his/her needs. As in case of any system development process where are number of short comings, there have been some short comings in the development of this system also.

7. FUTURE ENHANCEMENTS :

1) Adding pagination to the index page.

2) We can integrate the postgre SQL with Django.

3) Configuring of static assets can be done.

4) Integration of summer note WYSIWYG editor can be done.

5) Sitemap can be created.

8. REFERENCES :

M Alavi and P. Carlson, 1992. "A review of MIS research and disciplinary development," *Journal of Management Information Systems*, volume 8, number 4, pp. 45–63.

S. Albrecht, M. Lübcke, and R. Hartig–Perschke, 2007. "Weblog campaigning in the German Bundestag election 2005," *Social Science Computer Review*, volume 25, number 4, pp. 504–520.http://dx.doi.org/10.1177/0894439307305628

Arts & Humanities Citation Index, 2009. "*Arts & Humanities Citation Index*," at http://thomsonreuters.com/products_services/science/science_products/a-z/arts_humanities_citation_index/, accessed 2 September 2009.

E. Ashbee, 2003. "The Lott resignation, 'blogging' and American conservatism," *Political Quarterly*, volume 74, number 3, pp. 361–370.http://dx.doi.org/10.1111/1467-923X.00545

C. Auty, 2005. "U.K. elected representatives and their weblogs: First impressions," *Aslib Proceedings*, volume 57, number 4, pp. 338–355.http://dx.doi.org/10.1108/00012530510612077

J.R. Baker and S.M. Moore, 2008. "Blogging as a social tool: A psychosocial examination of the effects of blogging," *Cyberpsychology & Behavior*, volume 11, number 6, pp. 747–749.http://dx.doi.org/10.1089/cpb.2008.0053

J. Bar–Ilan, 2005. "Information hub blogs," *Journal of Information Science*, volume 31, number 4, pp. 297–307.http://dx.doi.org/10.1177/0165551505054175

A.F. Cameron and J. Webster, 2005. "Unintended consequences of emerging communication technologies: Instant messaging in the workplace," *Computers in Human Behavior*, volume 21, number 1, pp. 85–103.http://dx.doi.org/10.1016/j.chb.2003.12.001

W. Chen and R. Hirschheim, 2004. "A paradigmatic and methodological examination of information systems research from 1991 to 2001," *Information Systems Journal*, volume 14, number 3, pp. 197–235.http://dx.doi.org/10.1111/j.1365-2575.2004.00173.x

Python documents: https://docs.python.org/3.3/https://docs.python.org/3.3/install/index.html

Django : url: https://docs.djangoproject.com/en/3.0/intro/

Django quick installation : https://docs.djangoproject.com/en/3.0/intro/install/

https://djangocentral.com/building-a-blog-application-with-django/

https://djangocentral.com/how-to-a-create-virtual-environment-for-python/