TASK

Django - Blog App

Visit our website

Introduction

WELCOME TO THE DJANGO - BLOG APP TASK!

A blog is a platform for connecting and sharing information with the online community. This task will introduce you to the back-end logic required for setting up a basic blog on your website.

Disclaimer: we've curated the most relevant bits of the official documentation for you and added some additional explanation to make Django as concise and accessible as possible.

ADDING A BLOG APPLICATION

In this task you will add a blog to the hyperion project you created in the previous task. Recall the procedure for starting an application is as follows:

1. In your configuration directory (hyperion), open a command window and run the following command:

```
> python manage.py startapp blog
```

2. Install your app in **hyperion/settings.py**:

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

3. Setup a URL for your blog app in **hyperion/urls.py** as shown below:

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

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

There are a couple of important things to note in `urlpatterns`. Firstly, if you have a look at the path for our webapp app, you will see that it is slightly different to the one for our blog application. The reason for this is that we want our webapp application (which you created in the previous task) to be our home page i.e. when you startup your development server, we want our webapp to be at **http://127.0.0.1:8000/** and not **http://127.0.0.1:8000/webapp/**. Secondly, we have referenced the **urls.py** file in the blog, which we have to create because it doesn't exist yet.

Before creating the needed **urls.py** file for our blog, we need to create a model and a template that we will use in the blog application we are creating.

MODELS

Models are used to describe the data that must be stored for our app. Consider the blog app you are creating. This app will need to store the content of each blog post, information about when the blog post was created, and the author data related to a blog post. An example of a model for our blog post is shown below.

```
from django.db import models

# Create your models here.
class Post(models.Model):
   # Default behaviour - Django creates primary keys for you
   title = models.CharField(max_length=140)
   body = models.TextField()
   date = models.DateTimeField()

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

Notice that we are using object-oriented programming. We create a class called `Post`. The class describes all the properties (variables) and methods of each blog `Post` object. Add this code to your **blog/models.py** file.

With Django, each model generally maps to a single database table. Models contain metadata about your application's data. A database is a repository where you can store data that your app needs access to over a longer period of time. Variables, as you know, store data that your program needs to use and manipulate while the code is executing. However, when your program terminates, the data stored in variables are discarded. Sometimes you need to store data for longer. This data could be saved in files but accessing and manipulating data in files using code is not very efficient if there is a lot of data that needs to be stored. Databases store data in tables or other data structures that make data access and

manipulation using code much more efficient. Databases are therefore very important for web developers who create data-driven web applications.

Each class in **models.py** is a database table. Each attribute/variable of the model represents a database field. You specify a data type and maybe some further attributes (e.g. `max_length`) for each attribute. If you want to return data about your object, other than the fact that it is a `Post` or `Category` object, then you need to define the `__str__` method.

TEMPLATES

After we've defined our model, the next step is to create a template. **Django's** official documentation describes a template as follows: "A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted".

Inside your blog directory, create a new folder called **templates/**. Now, in **templates/**, insert an HTML file called **blog.html**. In **blog.html**, you should type the following using Django's template language:

```
{% extends "header.html" %}
{% block content %}
    {% for post in object_list %}
        <h5>{{ post.date|date:"Y-m-d" }}<a href="/blog/{{post.id}}">
{{ post.title }}</a></h5>
    {% endfor %}

{% endblock %}
```

In the example above, we are going to iterate through a list of blog objects. We'll display the `post.date`, in Y-M-D (year-month-day) fashion. We then specify a dynamic URL for the specific blog item so that a user can click the link to get more details about that particular blog. The URL will lead to **/blog/*<postid>***, and the text for the URL will be the title of the blog. We are then able to reference any of the elements we've defined in the **blog/models.py** file. An example of the output that will be displayed in the browser resulting from the template above (once we have completed our app and added some posts) is shown in the image below:

2022-08-19 Testing

2022-08-19 Hello Django!! Let's Blog

2022-08-19 Django Task 2

We also need to create a template for individual blog posts. Create **post.html** in **templates/** and insert the following code:

```
{% extends "header.html" %}
{% block content %}
<h3><a href="/blog/{{post.id}}">{{ post.title }}</a></h3>
<h6> on {{ post.date }}</h6>
<div class = "container">
    {{ post.body|safe|linebreaks}}
</div>
<br><br>
{% endblock %}
```

An example of the resulting output in the browser is shown below:

For more information about the template language used here, please consult the **Django documentation**. Notice that for both templates we have the following piece of code at the top:

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

This line of code is referencing a file called **header.html** which should be in the same directory as **post.html** or **blog.html** i.e. **templates/blog**. You'll notice that **header.html** doesn't exist, so you have to create it. Style your page however you like (remember that your blog posts will be displayed on this page). Alternatively, if you want your blog posts to be displayed elsewhere, such as your homepage, you can extend **index.html** which should be located in **personal/index.html** or **webapp/index.html**.

Now that we have a model for storing data related to our blogs and templates for displaying this data, we need to create the **urls.py** file mentioned earlier.

GENERIC DISPLAY VIEWS

Navigate to your blog app, create a new Python file called **urls.py**, and insert the following code:

```
from django.views.generic import ListView, DetailView
from django.urls import path
from .models import Post

urlpatterns = [
    path('',
        ListView.as_view(
            queryset=
            Post.objects.all().order_by("-date")[:25],
            template_name="blog.html"
            )
        ),
    path('<int:pk>/',
        DetailView.as_view(
            model = Post,
            template_name="post.html"
            )
        ),
]
```

Remember that **urls.py** is used to handle HTTP requests. This file specifies what resource to return when a certain HTTP request is made using a specific URL. Here, we are making use of one of Django's generic class-based view classes: `ListView`, which is used mainly for a webpage that presents a list of something to the user. In our case, the list is going to be the last 25 blog posts ordered by date, descending (hence the - (minus) sign). To access the data stored about our posts, we just reference our model. Because we already have the back-end database chosen in our settings, Django knows how to build the query in the background. Since Django builds the query for us, we don't have to use a specific query language (like SQL) to retrieve data from our database.

We also need to make use of another generic View, the `DetailView`. Individual blog posts will have their own page i.e. they will be located at **blog/1**, **blog/2** etc. Therefore, we need to create some sort of handling for this URL that will connect to a specific blog entry in the database and display the information. Thankfully, Django identifies each entry by its primary key. Each blog post object will be stored in a row in a database table. A primary key is a unique identifier for each row in a table in our database, therefore, no other entry should have the same primary key as another.

However, if you wanted to override the default behaviour, Django does allow you to do so relatively simply. Look at the example **models.py** file below and note the `id` line which sets the primary key.

```
class Post(models.Model):
   # Manually setting the primary key
   id = models.AutoField(primary_key=True)
   title = models.CharField(max_length=140)
   body = models.TextField()
   date = models.DateTimeField()

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

DATABASE MIGRATIONS

Whenever you define a new model in your models.py, you have to migrate it. Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. To do this, open your command window and run the following command in the configuration directory:

```
> python manage.py migrate
```

The next step is to do migrations specifically for our blog app:

```
> python manage.py makemigrations blog
```

Next, you have to setup your database table:

```
> python manage.py sqlmigrate blog 0001
```

Finally, you have to run migrate once again:

```
> python manage.py migrate
```

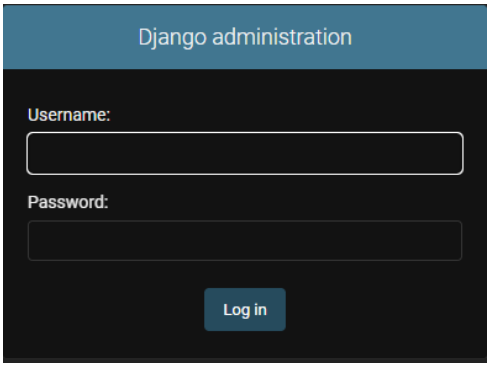
ADMIN

To add or edit the data in our database, we use Django Admin. To be able to do this, we need to modify the **blog/admin.py** file as shown below.

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

# Register your models here.
admin.site.register(Post)
```

To access the admin panel for your site, you can simply start up your development server, and go to **http://127.0.0.1:8000/admin/**.



However, you will be prompted for a username and password. To create your username and password, you have to return to your command window, and run the following command and follow the prompts thereafter:

Make sure to quit (CTRL + C) the development server before you run the command.

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

If you've followed the steps correctly, your blog should now be fully functional. Login to your admin panel with the credentials you just set up and you should see a table called Posts. Click on it and then click on 'add post' at the top right to add a blog post.

Remember that although the content of your posts can be of any character length, our post titles can be only 140 characters (including white spaces) long. This is because we've pre-defined the `max_length` of it as a parameter in our **models.py**. Feel free to edit your **models.py** should you require longer titles. If you make changes, don't forget to re-run your migration commands. You have to migrate every time you make changes to **models.py**.

Compulsory Task

If you have been accustomed to having a signature at the end of your email messages, you may want to do the same for your blog. A signature is a bit of text, like your contact information or a favourite quote, that is automatically added at the end of the messages you send.

Follow these steps:

- Edit your **models.py** and add a new field called `signature`. This field will follow the same syntax as your title:

```
signature = models.CharField(max_length=140)
```

- Place the field in-between your `body` and `date` fields.

- In addition to `max_length`, add a new parameter called `default` and set this to whatever you like (your name, favourite quote etc.). This should be a string.

- Make the necessary migrations.

- Add two blog posts.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCE

Django documentation. (n.d.). Django Software Foundation. Retrieved October 18, 2022, from **https://docs.djangoproject.com/en/4.1/**