

# **From the concept to the Cloud: Development of an Interactive Blog and Vlog Application using Django - Deployed with AWS.**

<http://3.93.175.171/>

**(Username: admin / Password: 1234)**

## **1. Introduction**

This report focuses on the development of an Interactive blog and vlog application using Django framework, which was later deployed by using AWS. The main propose of application is to allow users to create a blog and make posts, that include both textual and multimedia data. Additionally, there are other features such as commenting and liking posts that enhance user experience.

## **2. Preliminary**

### **Django Framework**

For the development of the application, the main tool used was Django, which is a high-level open-source python web framework. Django follows an MTV, model-view-template architecture. These components work together to facilitate rapid development and enhance code modularity. Additional benefits of the framework are the built-in features such as admin interface, authentication system, the built-in security features such as against SQL injection and the scalability and the versatility of the framework supporting both small and large applications.

### **Models**

Models are an essential component, representing the data structure of the application. Each model contains the essential fields, entities and behaviors of the storing data and they are automatically translated into database tables by Django, using the APIs, removing the hassle of writing, and creating raw SQL statements or quires for the developers. These models provide clean structured data making it easy to manage the information in the web applications.

### **Views**

Views in Django are simply python functions or classes that handle web requests such as HTTP requests and returns it as a web response, anything from HTML web responses, to images, redirection to another page or anything. It is an essential part of the user interface deciding how the data is presented to the users for consumption. Additionally, relationships between views and models are crucial since view obtains the necessary data from the models to generate the return responses.

## **Template**

Templates play an essential part in a Django framework. They consist of all the parts of the HTML static output, making it easier to input variables and display data from the views. This comes in handy in the management and maintenance of the code, making it easier for the developers.

## **Models – Views – Template- Relationship**

The relationship between these three are fundamental for the development of the application. Models define the data structure and create a database, these databases contain the information needed for the views to handle the logistic of the application and template focus on the end goal, by rendering the data received from the views into HTML, displaying at the user's browser. This collaboration forms the architecture of the Django framework which is the MTV architecture.

## **OOP implementation**

The Object-Oriented Programming method in Django provides a lot of benefits such as encapsulation of functionalities into classes which makes it easier for modularity and reusability. Additionally, it helps to organize and clean the code, making it easier to develop and maintain components such as models, views, and templates.

## **Migrations**

By implementing migrations, the database stays updated to all the recent changes such as insertion and deletion of a file. It also comes in handy, in the deployment process because of the version control. This makes the entire development process easier for the developers hence they can make any changes without affecting any other unrelated data.

## **AWS deployment**

Amazon Web Services works as a host machine making the Django web application available worldwide by utilizing its cloud infrastructure. AWS deploys the application by using services like EC2 for streamlines, RDS for managed databases and S3 to store media and static files. This comes with several benefits such as scalability, reliability, optimal performance security and more.

## **3. Project Overview**

The application is planned and develop in a way that it ensures ability and scalability. Every file is dedicated to something, and it has a purpose that provides a solid foundation to the functionality of the application. Additionally, the structure also follows OOP keeping everything organized and structured.

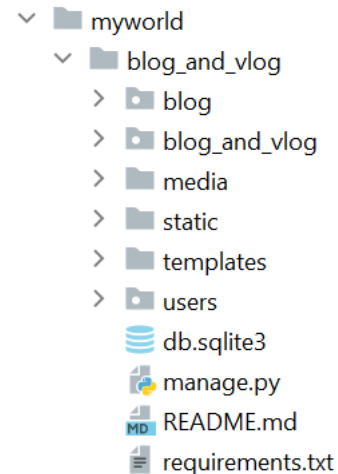
### **Apps:**

1. “blog\_and\_vlog” it's a core point for the project that ties numerous components together, defining how URLs are mapped and project-wide settings are configured. Additionally, it also stores all the installed apps.

2. “blog” that focuses on the blog related functionality and manages blog related data and user interactions.
3. “user”, handles all the data related to the user, as well as authentication, authorization and other.

### Folders:

5. “media”, stores all the images uploaded by the user. This facilitates the storage and retrieval of the media files.
6. “static”, stores all the static files and images. Ensuring consistent visual presentation through the application.
7. “templates”, contains all the HTML files, promoting the maintainability and reusability of the code, ensuring that the presentation layer doesn't mix with the logic of the application.
8. “db.sqlite3”, is the default SQLite database serving as the foundation of the application, created by Django to store all the structured data.



## 3. Project Implementation

### a) Project Setup

1. Started by setting up a virtual environment for the project, by using the following code: `python -m venv my world`. This enables efficient management of dependencies and prevents project conflicts.
2. Activated the environment: `source myworld/bin/activate`

(myworld) PS C:\Users\

3. Installing Django: `python -m pip install Django`
4. Started a Django project called `blog_and_vlog` : `django-admin startproject blog_and_vlog`, which automatically generates a project structure.

5. Creating the app called `blog`: `python manage.py startapp blog`
6. To complete the project setup, we need to Insert all apps created under `INSTALLED_APPS` in `settings.py` in the main project file `blog_and_vlog`.
7. Create migrations for changes made to the model: `python manage.py makemigrations`

```
blog/
  __init__.py
  admin.py
  apps.py
  migrations/
    __init__.py
  models.py
  tests.py
  views.py
```

```
blog_and_vlog
  __init__.py
  asgi.py
  settings.py
  urls.py
  wsgi.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
    'api.apps.ApiConfig',
    'users',
    # 'ckeditor',
]
```

8. Applies the changes to the model: `python manage.py migrate`
9. Start the development server `python manage.py runserver`
10. Ensure `http://127.0.0.1:8000/` is running.

## b) Model Creation

Models plays big part in the definition of the structure laying under the database. For example, each time a new instance is created, a new row corresponding to the same database is also created. In the model, there have been implemented thee python subclasses such as “Fields”, “Profile” and “Add\_comment”.

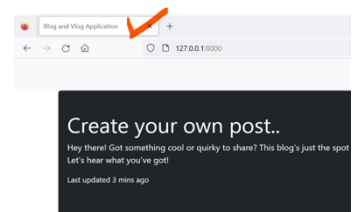
### "Fields" Model

This model contains all the essential information regarding the format and the structure of a blog post for a web application. The model contains all the pertinent data such as title, author, content, date, image, videos, and user interactions such as likes and comments. Furthermore, it keeps records the creator of the post and the number of likes it has received. This because the posts employ relationships like ForeignKey and ManyToManyField to link users and likes.

The following fields are defined in this model class:

Title: Identifies the blog post.

- Author: Represents the author's name, the creator of the post.
- AuthorHidden: A CharField allowing for hidden author information.
- (max\_length=255),
- Updated\_on: Automatically timestamps the post creation.
- Content: Stores the body content of the post that has turned into a text field in the Database.



- Created\_on: Has the same function as 'Updated\_on', to timestamps the post creation.
- Image and Video: Fields for uploading multimedia content along with the post.
- Like: ManyToManyField is used to establis a many-to-many relationship between a model, for user interaction such as liking a post.
- User: ForeignKey establishing a many-to-one relationship with the User model. This way, we are letting the application know that one user can write multiple posts. Additionally, it also contains functions like edit and delete that are fundamental for user interactions within the app and for consistent data management in the backend database.

```
# models.py
from django.db import models
class Fields(models.Model):
    title = models.CharField(max_length=255)
    author = models.CharField(max_length=255)
    authorHidden = models.CharField(max_length=255, null=True)
    updated_on = models.DateTimeField(default=timezone.now)
    content = models.TextField(blank=True, null=True)
    created_on = models.DateTimeField(default=timezone.now)
    image = models.ImageField(upload_to='images_uploaded', null=True)
    video = models.FileField(upload_to='videos_uploaded', null=True)
    like = models.ManyToManyField(User, related_name='blog_like')
    user = models.ForeignKey(User, on_delete=models.CASCADE, null=True)
    def total_likes(self):
        return self.like.count()
```

Methods:

1. `total_likes(self)` Calculates the total number of likes per post through the `ManyToManyField`.

## "Profile" Model

The 'Profile' model is an extension of the built-in 'User' model. It includes additional user personal information, including a biography, profile picture, and social media links.

- User: This has one to one relationship with the built in User model that ensures that each profile is linked to a single user instance. Note that if the user is deleted (`models.CASCADE`), then the associated instance is also automatically deleted.

- Bio: Text field stores the user's biography.

- Picture profile: Field for the user to add a profile picture.

- Facebook\_url,

- Twitter\_url,

- Instagram\_url:

Optional fields for storing social media profile URLs.

- Methods:

1. `def __str__(self): return str(self.user)`: for string representation. This is useful in the application, where model instances can be seamlessly identified by their username, help in the admin interface and user output.
2. `def get_absolute_url(self): return reverse('blog:home')`: for redirecting to the home page after performing an action on a particular page.

```
class Profile(models.Model):
    user = models.OneToOneField(User, null=True, on_delete=models.CASCADE)
    bio = models.TextField()
    picture_profile = models.ImageField(null=True, blank=True, upload_to='images/profile/')
    facebook_url = models.CharField(max_length=255, null=True, blank=True,)
    twitter_url = models.CharField(max_length=255, null=True, blank=True,)
    instagram_url = models.CharField(max_length=255, null=True, blank=True,)

    @ Chonthichar
    def __str__(self):
        return str(self.user)

    @ Chonthichar
    def get_absolute_url(self):
        return reverse('blog:home')
```

## "Add\_comment" Model

The 'Add\_comment' model is designed to handle comments on blog posts, linking each comment to a specific post through a foreign key.

- Post\_title: `ForeignKey` establishing a relationship with the 'Fields' model this ensures in the case of blog post removal all the associated comments are also automatically removed.

- Body: Field storing the content of the comment.

- Added\_date: Field indicating the date and time the comment was added.

```
class Add_comment(models.Model):
    post_title = models.ForeignKey(Fields, related_name="comments", on_delete=models.CASCADE)
    body = models.TextField()
    added_date = models.DateTimeField(auto_now_add=True)
    name = models.CharField(max_length=255)

    @ Chonthichar
    def __str__(self):
        return '%s - %s' % (self.post_title.title, self.name)
```

- Name: Field storing the user and their name when making a comment.
- Method:
  1. `'def __str__(self): return '%s - %s' % (self.post_title.title, self.name)'` : for string representation of the comment that included title of the post and user name of the commenter. This comes in handy while debugging, hence it provides clear description of each comment instance.

In the intricate orchestration of the Django web framework, Views, URLs, and Templates collaboratively shape the user experience. Here's an insight into their integration.

### c) Forms.py

Forms.py, helps with the user data management, data validation, as well as in the presentation of data on the user interface. Additionally, it makes the interactions between the front - end user interface and backend database easier. It does also have a similar structure to the models and the widgets variable and it is connected to “ModelForm” that directly connects to a form of a model.

```
class FillInForms(forms.ModelForm):
    # Chonhichar
    class Meta:
        model = Fields
        fields = ('title', 'author', 'authorHidden', 'updated_on', 'content', 'created_on', 'image', 'video')

    widgets = {
        'title': forms.TextInput(attrs={'class': 'form-control'}),
        # JS will automatically looks for value id
        'author': forms.TextInput(attrs={'class': 'form-control'}),
        'authorHidden': forms.TextInput(attrs={'class': 'form-control', 'value': '', 'id': 'name', 'type': 'hidden'}),
        'updated_on': forms.DateInput(attrs={'class': 'form-control', 'type': 'date'}),
        'content': forms.Textarea(attrs={'class': 'form-control'}),
        'created_on': forms.DateInput(attrs={'class': 'form-control', 'type': 'date'}),
        'user': forms.Select(attrs={'class': 'form-control'}),
        'image': forms.ClearableFileInput(attrs={'class': 'form-control'}),
        'video': forms.ClearableFileInput(attrs={'class': 'form-control'}),
    }

    # Chonhichar
    def get_object(self, queryset=None):
        title_id = self.kwargs.get('title_id')
        return get_object_or_404(Fields, pk=title_id)
```

### I. 'FillInForms'

All the data regarding the blog posts is managed by 'FillInForms' and is linked to a 'ModelForm,' that is designed for creating and editing of instances that lay within the 'Fields' model. Advantage of this is that it makes it easier to render form elements such as title and author, with the help of bootstrap styling “form-control”. Additionally, it does also include a code, that retrieves a blog posts for editing.

- **class FillInForms(forms.ModelForm):**: Automatically generates fields linked to Fields' model attributes.
- **model = Fields:** links the form to the 'Fields' model.
- **fields = ():** specifies included model fields in the form.
- **widgets = {}:** clarifies the HTML widgets for each field, the 'attr' dictionary in each widget sets HTML class like id and type.
- **def get\_object:** Fetches a specific blog post of the 'Fields' model based on the title 'ID'.

```
class FillInForms(forms.ModelForm):
    # Chonhichar
    class Meta:
        model = Fields
        fields = ('title', 'author', 'authorHidden', 'updated_on', 'content', 'created_on', 'image', 'video')

    widgets = {
        'title': forms.TextInput(attrs={'class': 'form-control'}),
        # JS will automatically looks for value id
        'author': forms.TextInput(attrs={'class': 'form-control'}),
        'authorHidden': forms.TextInput(attrs={'class': 'form-control', 'value': '', 'id': 'name', 'type': 'hidden'}),
        'updated_on': forms.DateInput(attrs={'class': 'form-control', 'type': 'date'}),
        'content': forms.Textarea(attrs={'class': 'form-control'}),
        'created_on': forms.DateInput(attrs={'class': 'form-control', 'type': 'date'}),
        'user': forms.Select(attrs={'class': 'form-control'}),
        'image': forms.ClearableFileInput(attrs={'class': 'form-control'}),
        'video': forms.ClearableFileInput(attrs={'class': 'form-control'}),
    }

    # Chonhichar
    def get_object(self, queryset=None):
        title_id = self.kwargs.get('title_id')
        return get_object_or_404(Fields, pk=title_id)
```

## II. ‘EditForms’

Allows the users to edit contents in the “Fields” mode. The users can edit their blog posts such as changing the title, content, or any related multimedia such as pictures and videos.

- **class**

```
class EditForms(forms.ModelForm):
    class Meta:
        model = Fields
        fields = ('title', 'updated_on', 'content', 'image', 'video')

        widgets = {
            'title': forms.TextInput(attrs={'class': 'form-control', 'placeholder': 'This is Title place holder.'}),
            'updated_on': forms.DateInput(attrs={'class': 'form-control', 'type': 'date'}),
            'content': forms.Textarea(attrs={'class': 'form-control'}),
            'image': forms.ClearableFileInput(attrs={'class': 'form-control'}),
            'video': forms.ClearableFileInput(attrs={'class': 'form-control'}),
        }
```

**EditForms(forms.ModelForm):** Creates a new form class for editing that inherits from a Django Model Form.

- **class Meta:** Comes from the properties configuration.
- **model = Fields:** Connects the form to the ‘Fields’ model.

```
class AddCommentForm(forms.ModelForm):
    class Meta:
        model = Add_comment
        fields = ('name', 'body')

        widgets = {
            'name': forms.TextInput(attrs={'class': 'form-control'}),
            'body': forms.Textarea(attrs={'class': 'form-control'}),
        }
```

- **fields = ():** Specifies fields from the ‘Field’ model included in the form.
- **widgets = {}:** clarifies HTML widgets for each field and the ‘attr’ dictionary sets HTML class like ‘id’ and ‘type’.

## III. ‘AddCommentForm’

The ‘AddCommentForm’ is the form class connected to the class ‘Add\_comment’ in ‘model.py’, helps users to submit their name and comments within a web application. It is created to render the ‘name’ and ‘body’ fields with user-friendly input areas, styled using Bootstrap’s ‘form-control’ class for a clean and nice look on the webpage.

- **class AddCommentForm(forms.ModelForm):** Showcases form’s connection to the model
- **class Meta:** Assists the form with the settings by instructing Django on which model to use and what fields to include when creating comment form.
- **model = Add\_comment:** Linked to ‘Add\_comment’ in form.
- **fields = ():** Instructing Django on which field it should include the form. In this case, we want to display only (name, body)
- **widgets = {}:** The way of rendering HTML form elements. Custom widgets for the fields mentioned in the code are defined by this dictionary.

### d) Implementation of Views, URLs and Templates

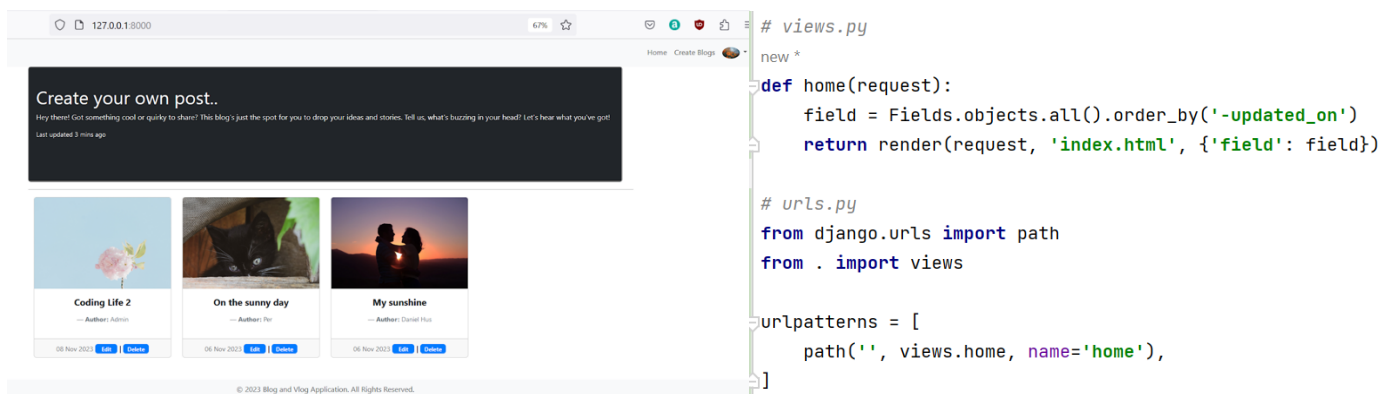
Views, URLs and Templates play a critical role in a Django Framework. They work together to provide a great user experience for the users. Here’s an insight into their integration:

## Views

Views are Python functions that receive a web request and return them into a web response. The URLs map these requests to their corresponding views. In the 'blog\_and\_vlog' project, there are multiple apps in the 'view.py', that are defined in 'blog' app and 'user' app based on its specific use. They contain the logic to handle requests, interact with models to fetch and save data to templates.

### I. Views.py in blog application

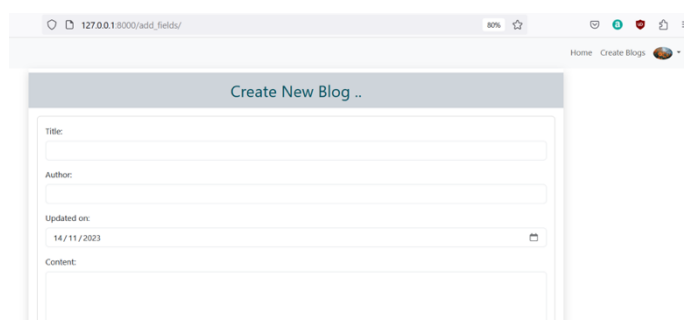
Home View: Connects the root URL ("") to the home view. This ensures the display of index.html template with all the blog articles are presented in a chronological order.



Details View: Is accessible via a URL art/int:title\_id and it displays a specific post's (using title\_id) details, including likes, and sends the data further to the 'detail.html' template.



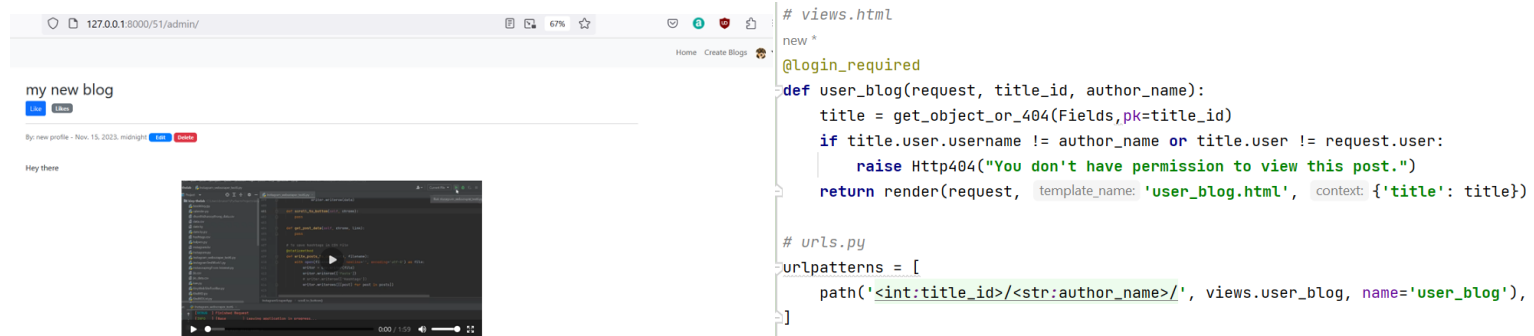
Fill\_in\_form View: This field is protected by the @login\_required, which makes it only possible to create a post, after the user has logging in to their account. Thereafter, the 'fill\_in\_form' view maps to 'add\_fields' to show the page/ form



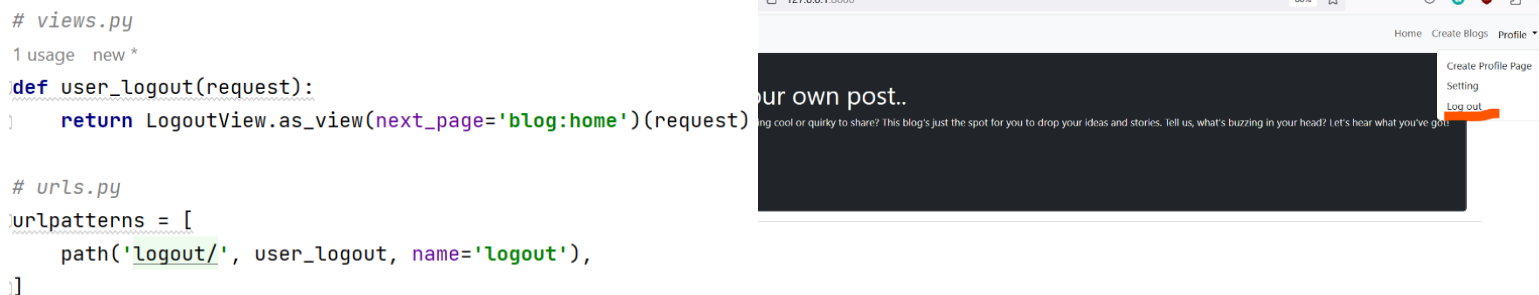


where the users can create a new post. After creating the post, it redirects the users to their blog page.

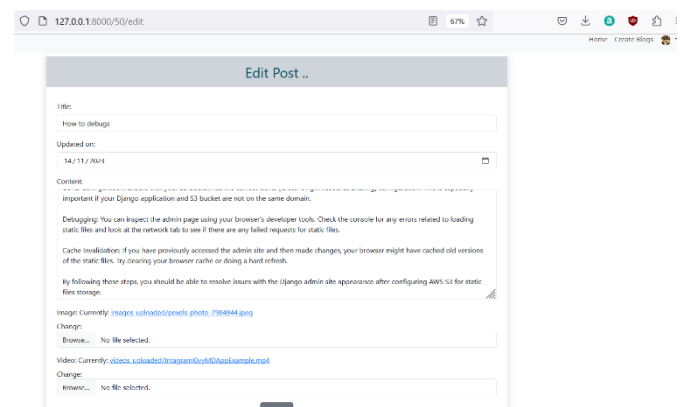
User\_blog View: Authenticates users and checks ownership before displaying the blog using 'user\_blog.html' accessible via <int:title\_id>/<str:author\_name>. It will display, an error message if the user is not the owner of the blog.



User\_logout: Logs out users using the built-in 'LogOutView' and redirects to the home page.



EditPostView View: This is an UpdateView, that allows users to edit their blog post which can be displayed from 'update\_blog.html' template, thereafter redirects user to the post's detail view.



```

class EditPostView(UpdateView):
    model = Fields
    form_class = EditForms
    template_name = 'update_blog.html'

# views.py
1 usage new *
class DeletePostView(DeleteView):
    model = Fields
    template_name = 'delete_blog.html'

new *
def get_success_url(self):
    return reverse('blog:home') #

# urls.py
urlpatterns = [
    path('<int:pk>/delete', DeletePostView.as_view(), name='delete_post'),
]

```

DeletePostView View: Deletes a post and redirects user to the home page.



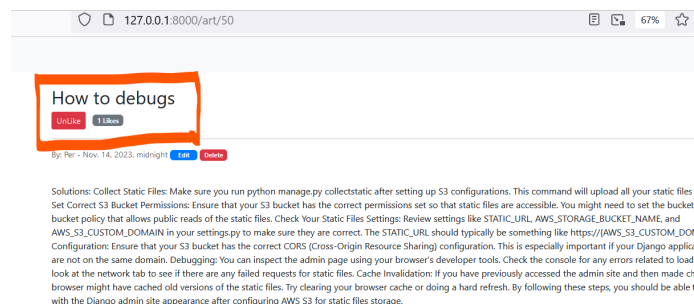
LikeView View: Tracks likes for a post and redirects to the post's detail view.

```

# views.py
1 usage new *
def LikeView(request, pk):
    title = get_object_or_404(Fields, id=request.POST.get('title_id'))
    if title.like.filter(id=request.user.id).exists():
        title.like.remove(request.user)
    else:
        title.like.add(request.user)
    return HttpResponseRedirect(reverse('blog:detail', args=[str(pk)]))

# urls.py
urlpatterns = [
    path('<int:pk>/like', LikeView, name='like_post'),
]

```



AddCommentpage View: 'CreateView', Handles comment creation with 'Add\_comment' and saves it by using 'AddCommentForm' and redirects to the homepage.

```

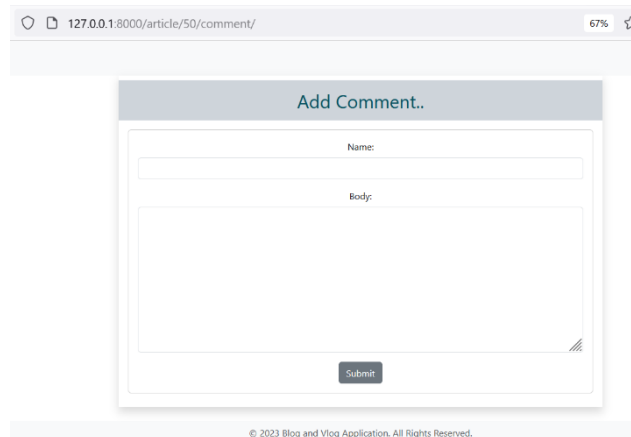
# views.py
1 usage new *
class AddCommentpage(CreateView):
    model = Add_comment
    form_class = AddCommentForm
    template_name = 'post_comment.html'

new *
def form_valid(self, form):
    form.instance.post_title_id = self.kwargs['pk']

    return super().form_valid(form)
    success_url = reverse_lazy('blog:home')

# urls.py
urlpatterns = [
    path('<int:pk>/comment', AddCommentpage.as_view(), name='add_comment'),
]

```



## II. Views.py in user application

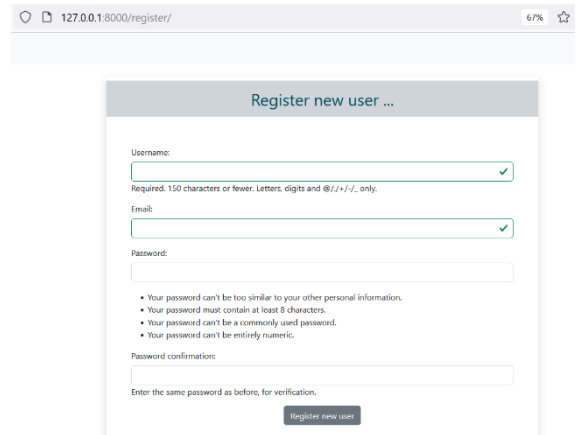
In the user applications the views.py includes several view functions and classes that manage different aspects of user interaction, from registration to profile customization.

UserRegisterPageView: Manages user registration using 'SignUpForm' in template 'registration.html' and redirects post-registration if successful.

ProfilePageView: Renders user profile details from 'profile\_page.html' templates, based on the user's unique ID.

```
# views.py
1 usage new *
class UserRegisterPageView(generic.CreateView):
    form_class = SignUpForm
    template_name = 'registration/registration.html'
    success_url = reverse_lazy('login')

# urls.py
urlpatterns = [
    path('register/', UserRegisterPageView.as_view(), name='register'),
]
```



EditProfilePage: Allows users to edit their profile form and displays the changes by using 'edit\_profile\_page

\_for\_user.html' and redirects them to the homepage after a successful submission.

```
# views.py
1 usage new *
class EditProfilePage(generic.UpdateView):
    model = Profile
    template_name = 'registration/edit_profile_page_for_user.html'
    fields = ['bio', 'picture_profile', 'facebook_url', 'twitter_u
    success_url = reverse_lazy('blog:home')

# urls.py
urlpatterns = [
    path('user/<int:pk>/edit_profile_page/', EditProfilePage.as_v
]
```

```
# views.py
1 usage new *
class CreateProfilePageForUser(CreateView):
    model = Profile
    form_class = PageForm
    template_name = "registration/create_profile_page_for_user.html"
    new *
    def form_valid(self, form):
        form.instance.user = self.request.user
        return super().form_valid(form)

# urls.py
urlpatterns = [
    path('create_profile/', CreateProfilePageForUser.as_view(), name='create_profile_page'),
]
```

CreateProfilePageForUser: Creates a user's profile page using 'PageForm' and 'create\_profile\_page\_for\_user.html' template, linking it to the current user.

LogoutView: Handles user logout and redirects to the homepage, in additionally to ensure that a profile bring created is linked to the user logged, the form valid is overriding in the form valid method in the 'CreateProfilePageForUser'.

```
# urls.py
urlpatterns = [
    path('logout/', LogoutView.as_view(), name='logout'),
]
```

This line of code ensures that the user that is currently logged-in, matches with the ID of the author of the post, ensuring that only the creator of the post is able to make modifications to it.

```
{#      current user:  {{ user.id }}#}
{#      blog author:  {{ title.user.id }}#}
{#      <p>user page {{ page_user }}</p>#}
```

## e) URLS Configurations

**Blog App:** In the 'urls.py' file of the Blog app, URL patterns define endpoints for various views, ensuring incoming requests are directed to the appropriate view functions for response handling. This organization is detailed in Section 3.5 A: Views.

```
app_name = 'blog'
urlpatterns = [
    path('', views.home, name='home'),
    path('art/<int:title_id>', views.detail, name='detail'), # add pots
    path('add_fields/', views.fill_in_form, name='add_fields'), # add pots
    path('<int:title_id>/<str:author_name>', views.user_blog, name='user_blog'),
    path('accounts/', include('django.contrib.auth.urls')),
    path('logout/', LogoutView.as_view(), name='logout'),
    path('<int:pk>/edit', EditPostView.as_view(), name='edit_post'), # Edit
    path('delete/<int:pk>', DeletePostView.as_view(), name='delete_blog'),
    path('like/<int:pk>', LikeView, name='like_post'),
    path('article/<int:pk>/comment/', AddCommentpage.as_view(), name='post_comment'),
]

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

```
# urls.py
urlpatterns = [
    path('register/', UserRegisterPageView.as_view(), name='register'),
    path('edit_profile/', UserEditPageView.as_view(), name='edit_profile'),
    path('user/<int:pk>/profile/', ProfilePageView.as_view(), name='profile_page'),
    path('user/<int:pk>/edit_profile_page/', EditProfilePage.as_view(), name='edit_profile_page'),
    path('create_profile/', CreateProfilePageForUser.as_view(), name='create_profile_page'),
]

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

**User Application:** The URL patterns within the 'urls.py' file of the User app, are important for the configuration and management of the user authentication and profile features.

**Url in settings:** In the main project structure, the 'urlpatterns' list in the 'urls.py' file within the 'settings' directory serves as a central registry for all URL patterns across the 'blog and vlog' app. Key inclusions are:

- **'path('admin/')':** Connects to the admin page, granting site administrators application control.
- **'Include('blog.urls')':** Integrates URL patterns from the blog app and Django's built-in authentication system.
- **'Include('users.urls')':** Integrates authentication views for user login, logout, and profile.
- The **'if settings.DEBUG'** condition facilitates serving media files during development, emphasizing its inappropriateness for a production environment.

```
# urls.py
urlpatterns = [
    path('', include('blog.urls')),
    path('admin/', admin.site.urls),
    path('accounts/', include('django.contrib.auth.urls')),
    path('users/', include('django.contrib.auth.urls')),
    path('', include('users.urls')),
]

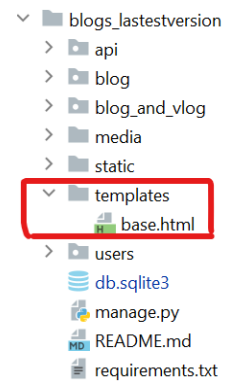
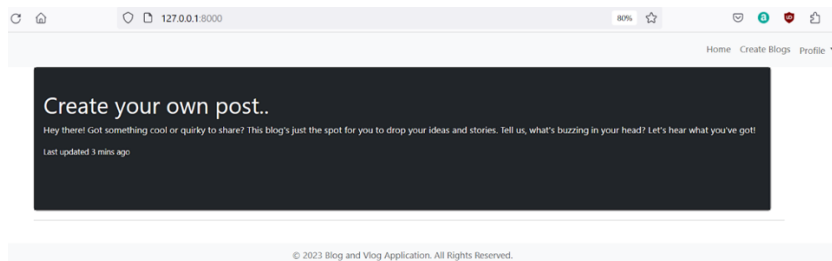
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

## f) Templates

For each template file, there is a corresponding view, which is kept in the template's directory within each app. They are defined as global templates, called `base.html`, in a blog application. Each of them, match the views and URLs that are described in section 3.5.A: Views. Additionally, there is also overview of the global and subsidiary templates within the application.

### I. Global template: The “base.html”

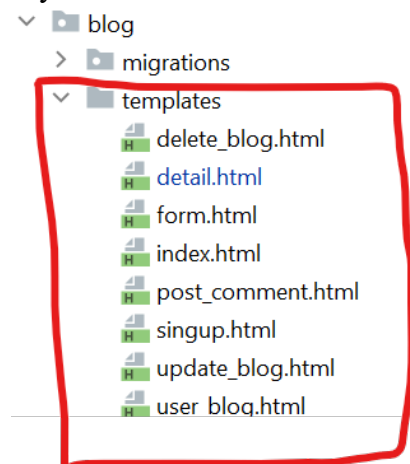
The global template, ‘base.html’ functions as template for the entire project and it is designed to create a cohesive and user-friendly interface. It incorporates HTML head elements, navigation bars, main content blocks, and footers with the help of Bootstrap for responsive design, that ensures a consistent look and feel across all pages.



### II. Subsidiary Templates

Within the 'templates' directory, each template file corresponds to a specific view within the Blog application. These templates, inheriting from 'base.html,' maintain a consistent layout and style. Key templates include:

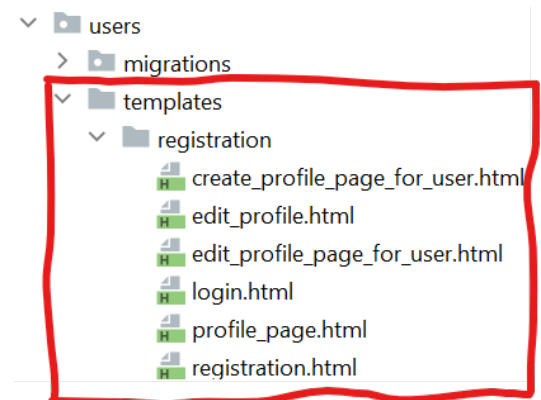
- 'Delete\_blog.html': A confirmation blog for deleting posts by authenticated users.
- 'Detail.html': Displays details of a specific blog post, including users to like, unlike, edit, delete, if registered and comment section to even unregistered users.
- 'Form.html': Facilitates the creation of new blog posts for authenticated users, including form fields for title, content, images, and videos.
- 'Index.html': Serves as a summary landing page, listing all blog posts.
- 'Post\_comment.html': Provides a form section for any user enabling them to comment on posts.
- 'Signup.html': Contains the registration form for users to join the app.
- 'Update\_blog.html': Enables authenticated users to edit their own blog posts.
- 'User\_blog.html': Displays individual blog posts of the user.



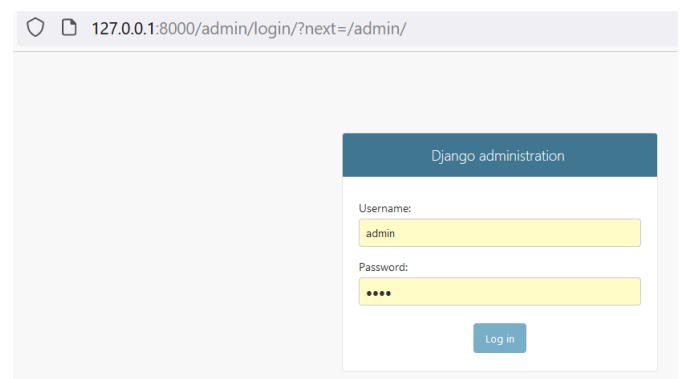
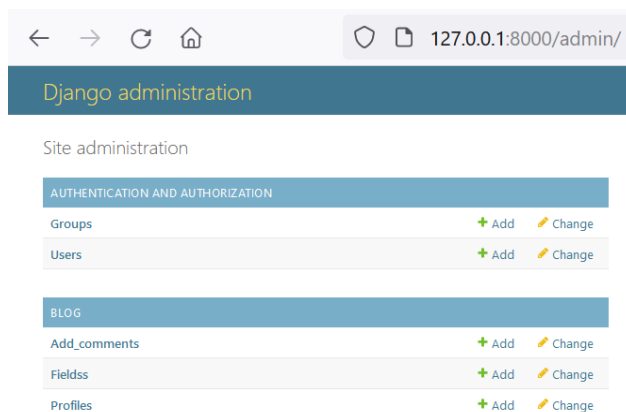
### III. Users

The user application within the blog and vlog application, contains a 'templates' directory which includes a 'registration' subdirectory that contains HTML files associates to the user registration and profile management:

- 'Create\_profile\_page\_for\_user.html': Used to create a profile when a user logs into the app for the very first time.
- 'Editing\_profile.html': Allows the user to modify their personal info on the settings page such as username, email address and the password.
- 'Edit\_profile\_page\_for\_html': Allows user to modify information on their profile page such as bio, image profile and others.
- 'Login.html': Displays the login form for both the users that are registered and the ones that have submitted their registration.
- 'Profile\_page\_html' \_ Displys the user profile page.
- 'Registration.html': Registers the new user on the application.



#### g) User authentication and admin interface.



The admin interface in Django is a powerful tool for the management of the data model of an application (admin page can be accessed by username: admin and password: 1234). To employ it in the 'blog' app, models must be registered within the 'admin.py' file in the 'blog' app, authorizing the admin to to create, read, delete, update operation from the panel as shown in screenshot.

```
1 usage  Chonthichar *
class FieldsAdmin(admin.ModelAdmin):
    list_display = ('id', 'title', 'author', 'updated_on',
                   'content', 'created_on', 'image', 'video', 'user')

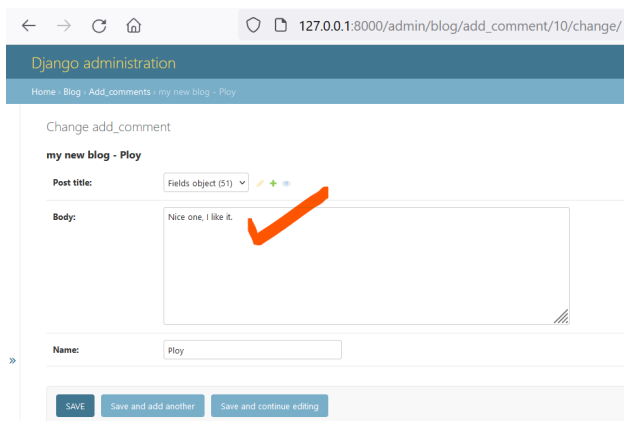
# class FiledProfile(admin.ModelAdmin):
#     list_display = ('id', 'user')

admin.site.register(Fields, FieldsAdmin)
admin.site.register(Profile)
admin.site.register(Add_comment)
```

Furthermore, the user is able add multimedia their posts and also manage comments, for this, we must add we must add specific to be displayed, in the nice format by defending 'FieldsAdmin' class and settings attributes like 'list\_display', in the admin list.

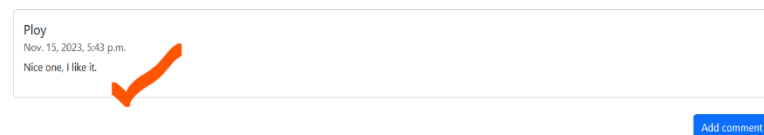


Additionally, anyone may leave a comment without an username, making the app open and user-friendly. Everyone to able to express their opinion, promoting and boosting app usage. app usage.



## Comments..

1 Comment(s)



## 5. Implementation of OOP in the application.

### a) 'Model.py' in 'blog\_and\_vlog

1. **Inheritance:** Django provides a built-in class model called "models.Model" that is inherited by all classes (Fields, Profile, Add comment). These include all the functionalities such as database, query and others automatically.
2. **Encapsulation:** Every class has it's own unique behaviour and data. For example, it has classes that includes all the data, regarding the blog posts, such as a post that has been saved and also the mount of likes and coments in the post.
3. **Composition:** Serveral field objects such as 'models.ImageField' and 'models.ForeignKey' are used to construct the models in the model code, where each of them contributes their functionality to the model, this way the code that handles the file storage and dabase relations dosen't needs to be weitten by hand. Allowing the 'model.py' handle more challenging taks while maintaining the the code clean and modular.

### b) 'Admin.py' in the blog app

In the Admin.py, there are models like Fields, Profile and Add\_comment that are registered for the admin interface The FieldsAdmin class is used to customize the visualization of the h 'Fields' in the admin panel. This ensures consistent data management with OOP principals for Create, read, update and delete

```
# admin.py
1 usage Chonthichar *
class FieldsAdmin(admin.ModelAdmin):
    list_display = ('id', 'title', 'author', 'updated_on',
                  'content', 'created_on', 'image', 'video', 'user')

admin.site.register(Fields, FieldsAdmin)
admin.site.register(Profile)
admin.site.register(Add_comment)
```



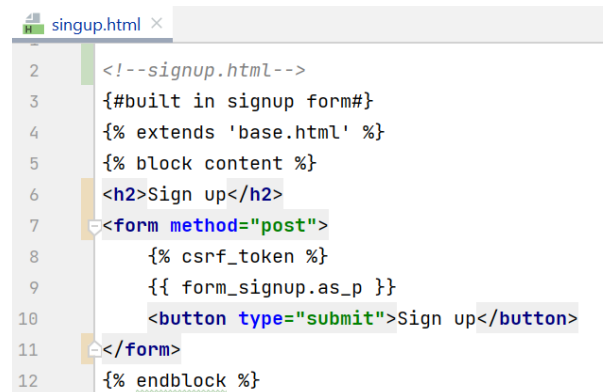
### c) 'Forms.py' in blog\_and\_vlog application

- Inheritance: Every form class such as 'FillInForms', 'EditForms', 'AddCommentForm' are subclasses of 'forms.ModelForm' which again is a subclass of 'forms.Form', meaning that all the functionality is automatically inherited by each form class. This is a basic idea of OOP since each subclass in a model inherits methods and properties from their parent class.
- Encapsulation: It is clear how each form class ('FillInForms', 'EditForms', 'AddCommentForm') hold their own logic and data by containing everything needed in the functions like fields, widgets, 'get\_object' within a single class. This corresponds to the idea that all an encapsulation needs are contained within.
- Abstraction: Makes it easier to understand the complexities of HTML creating and form input by handling and abstraction the code. An example to this is when we use Django's 'forms' instead of looking into the specifics of HTML form elements. By using 'forms.TextInput', Django automatically creates an HTML <input type="text">ML on the webpage, making the development quick and easy.

### d) Django Templates

The OOP concepts are implemented in the app templates where the `{% extends 'base.html' %}` tag is used to implement inheritance enabling templates to share a common layout and style. Encapsulation is showcased in the html file by how the data is presented in a clean and organized manner for example in the `signup.html`.

Polymorphism is shown in the usage of `{% block %}` tags by creating a space where child templates can add their own custom content. Additionally, each child template has their own content that illustrates the concepts of methods having different forms.



```
1  <!-- signup.html -->
2
3  {% built in signup form %}
4  {% extends 'base.html' %}
5  {% block content %}
6      <h2>Sign up</h2>
7      <form method="post">
8          {% csrf_token %}
9          {{ form_signup.as_p }}
10         <button type="submit">Sign up</button>
11     </form>
12     {% endblock %}
```

```
{% extends 'base.html' %}
{% block title %}Create a new blog post {% endblock %}
```

## 6. Database Migrations

Migration is a method to implement the changes into the model while guaranteeing that any new features and updates keep the website running smoothly and functional, up to date. Migrations have to be implemented whenever new features are added, removed or modified, facilitating the developers work and enhancing the website. Other developers can update their own database by running the migrations on their own machine without changing the primary database schema.



## 1. Migration Process:

When the structures of the models have been decided and the changes have been made accordingly, it is time to create migrations by implying `python manage.py makemigrations`. Thereafter, the `python manage.py migrate` implementation has to be made to ensure the changes. This allows the changes to be reviewed before affecting the database.

The default SQLite database, provided by Django, is helpful in the project testing step. However, it is not suitable for the production because of the issues regarding scalability and durability. SQLite3 must also be converted to PostgreSQL, hence AWS does not support it, in contrast to Django.

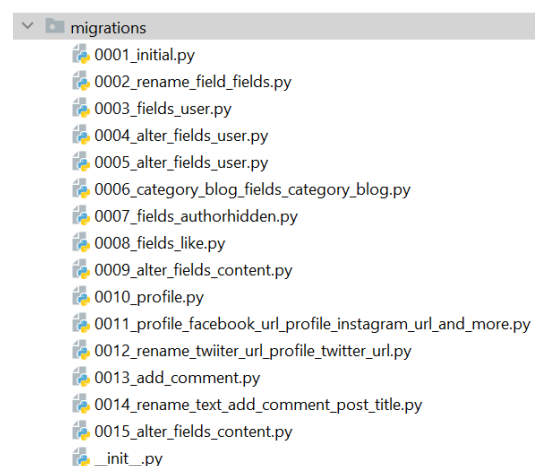
After implementing the changes, they are reflected in the project's 'Databases' configurations.

Additionally, migrations must be implemented to establish the schema in the new database.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'backendcourse1',
        'USER': 'mysuperuser',
        'PASSWORD': 'mysuperuser',
        'HOST': 'backendcourse1.c7xdmkaitlp.us-east-1.rds.amazonaws.com',
        'PORT': '5432',
    }
}
```

## 2. Screenshot and output:

The history of schema modifications is showcased in the 'blog\_and\_vlog' project's migrations list. Migration files such as, `0001_initial.py` and `0015_alter_fields_content.py` showcase the number of changes made along the project. Each migration file is an incremental step that modifies the database schema to match the current state of models. This system of version control for databases is crucial for collaborative development and smooth deployment to production servers.



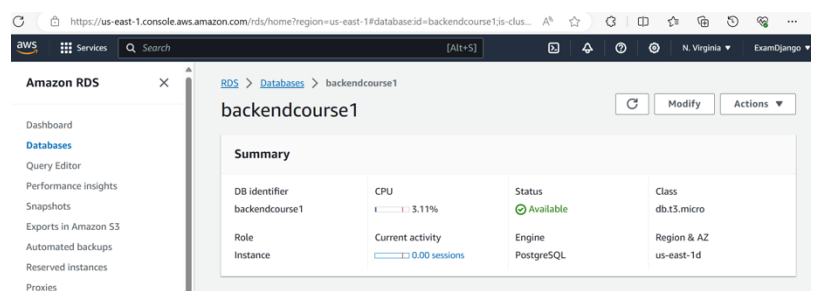
## 7. AWS Deployment

The page is accessible by clicking in the following link: [Blog and Vlog Application](#)

For the deployment of the application to the cloud we followed three steps.

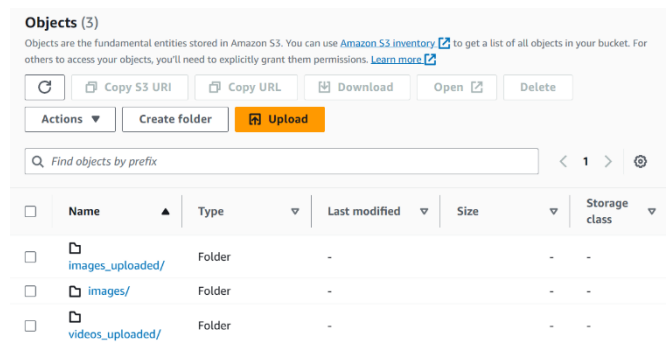
### a) RDS Instant Launch:

Migrating from SQLite3 to PostgreSQL. This way the app is configured to use an AWS RDS instance by modifying the database in the Django configuration.



## b) S3 Bucket Setup:

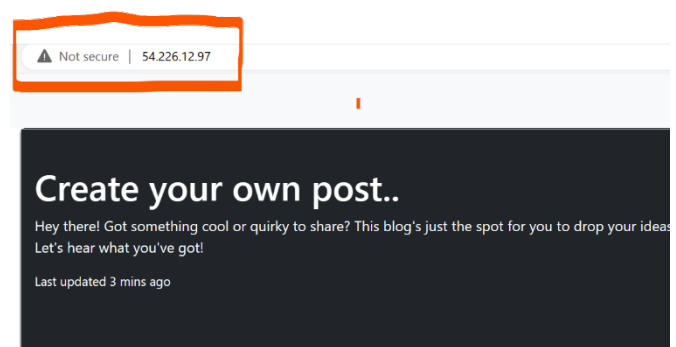
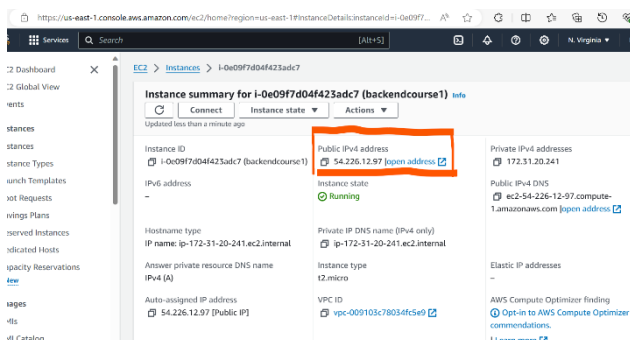
By setting up an S3 (Simple storage service) bucket on AWS for managing static and media files allowed the users to upload videos and images from anywhere on the web. All the data is stored on the cloud storage service, ensuring accessibility and scalability.



## c) Application deployment on AWS EC2:

Even with AWS offering a wide range of services for Django project deployment, it is not suitable for the production environment due to its scalability limits on the Django development server. To solve this, Gunicorn was installed for the efficiency of scaling and Nginx as a reverse proxy to serve static file. This setup was manually configured in a Linux environment.

```
AWS_ACCESS_KEY_ID = 'AKIAYNGRWB5GZB6RBYFK'  
AWS_SECRET_ACCESS_KEY = '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'  
AWS_STORAGE_BUCKET_NAME = 'backendcourse1'  
AWS_S3_SIGNATURE_NAME = 's3v4',  
AWS_S3_REGION_NAME = 'us-east-1'  
AWS_S3_FILE_OVERWRITE = False  
AWS_DEFAULT_ACL = None  
AWS_S3_VERIFY = True  
DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
```



## 8. Challenges

Several issues were encountered when deploying the blog to the cloud. The first issue, related to the unexpected costs surged when deploying the application on AWS Elastic Beanstalk, to solve this we opted to switch to AWS EC2 to get more control and flexibility, however another issue surges when setting up a live server on EC2 which was ultimately solved by configuring Nginx as a server proxy and web server, which also changes from SQLite to PostgreSQL for database management.

Static and media file hosting was another area of great struggle. Since Django does not come with the function to handle static and media files in production by default, there was a need to implement white noise to serve a production correctly. Another challenge was to connect similar data across different pages, to be able to solve this, Key(PK) based relationships were implemented to guarantee data consistency and connection between the sites.

In conclusion, these challenges provided us with valuable lessons regarding how to overcome technical difficulties such as file management in Django and as well as, deployment challenges. More importantly, we learned that the key to overcome obstacles often lies in the curiosity to adapt and explore alternative solution or strategies, especially when it does not go as planned.

## 9. Conclusion

In conclusion, despite of the encounter of several obstacles along the process, the end goal of development of Django project and deployment of the application on AWS was achieved. This project represents our ability to overcome challenges such as intricacies with the database relationships, static file management and server configuration for creating of an application that runs smoothly and has a user-friendly interface, including features like login, post, like, commenting in addition to the key features such as profile management system.

Furthermore, the 'Blog and Vlog' application has potential for further development hence it provides a dynamic framework for potential feature enhancements integrations and expansions.

## 10. References

- [www.w3schools.com](https://www.w3schools.com/django/). (n.d.). Django Tutorial. [online] Available at: <https://www.w3schools.com/django/>
- MDN Web Docs. (n.d.). Django introduction. [online] Available at: <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction>
- [docs.aws.amazon.com](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-django.html). (n.d.). Deploying a Django application to Elastic Beanstalk - AWS Elastic Beanstalk. [online] Available at: <https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-django.html>
- Otto, M. (2022). Bootstrap. [online] Getbootstrap.com. Available at: <https://getbootstrap.com/>
- Django Project. (n.d.). Django. [online] Available at: <https://docs.djangoproject.com/en/4.2/topics/migrations/>
- Django Documentation Release 3.0.15.dev Django Software Foundation. (2021). Available at: <https://buildmedia.readthedocs.org/media/pdf/django/3.0.x/django.pdf>
- [www.youtube.com](https://www.youtube.com/watch?v=7O1H9kr1CsA). (n.d.). Deploy a Django web app with Nginx to Amazon EC2. [online] Available at: <https://www.youtube.com/watch?v=7O1H9kr1CsA>
- Otto, M. (2022). Bootstrap. [online] Getbootstrap.com. Available at: <https://getbootstrap.com/>
- Django Documentation Release 3.0.15.dev Django Software Foundation. (2021). Available at: <https://buildmedia.readthedocs.org/media/pdf/django/3.0.x/django.pdf>
- [docs.aws.amazon.com](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-django.html). (n.d.). Deploying a Django application to Elastic Beanstalk - AWS Elastic Beanstalk. [online] Available at: <https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-django.html>
- Deploy Django Application on EC2 with PostgreSQL, S3, Domain, and SSL Setup. [online] Available at: <https://www.codewithmuh.com/blog/deploy-django-application-on-ec2-with-postgresql-s3-domain-and-ssl-setup>