Post – It

(Code Review Documentation)

Name – Pritish Gupta

Email – pritishgupta009@gmail.com

Link to project - https://github.com/PRITISH009/Post-It

Organization – Global Alliance for Genomics and Health

Project Name - Web application for manual curation of trait-to-ontology mappings

Mentor(s) - Kirill Tsukanov, Timothee Cezard

March 2020

Contents

Sections **Page No.**

1. Introduction …………………………………………………………………… 03
2. Steps to Set Up App on your machine ……………………………… 04
   I. Requirements ………………………………………………………… 04
   II. Setup ……………………………………………………………………… 05
3. Functionalities Explained (Application Tutorial) ………………… 06
4. Code Documentation ……………………………………………………. 11
   I. General Over of Project directory ………………………………….. 11
   II. Routing in the Project ………………………………………………... 12
   III. Working Mechanism of the Project ……………………………… 14
   IV. System Workflow …………………………………………………… 15
   V. Web Interface through Class Based Views ……………………. 16
   VI. Models/Database Schema of the Project ……………………… 23
   VII. Templates ……………………………………………………………. 27

Introduction

“Post – It” is a Django based application designed to help you get ahead of your day. The main purpose of developing such a simple yet efficient application is to improve the productivity of user in daily life. This app aims to provide a platform for managing all of user’s daily task.

The application provides the user with a personalized wall where the user can post all of their tasks that he/she wants to complete. The application also consists of Login facilities which enables the application to provide a customizable profile to the user.

A user can also update and delete a Note posted on the wall as well. A user can also customize his/her profile by updating his/her Username, Email linked to the account and can also upload a Profile picture.

Steps to Setup the app on your own machine

Requirements –

There are a few things that has to be done before you can successfully run the app on your own machine.

1. First of all, your machine should have Python 3 installed in it but any version of Python 3.7 might not allow you to access admin page of the app (There is some issue regarding this and the only easy solution I found was to upgrade your Python from 3.7 to 3.8. So, it is strongly recommended to upgrade your Python version to any variant of Python 3.8. You can download Python 3.8 from here - https://www.python.org/downloads/. And don't forget to add your python to Path variable so that it (Python) is callable from Terminal or PowerShell or any similar environment.
2. This is optional. If you already have a Terminal on your system then you can skip this step but for those on Windows, I recommend using Cmder from here - https://cmder.net/. This very similar to a Unix based Terminal and give a ton of facilities. Installing this is totally up to the user. Download and Install Cmder from the above link. Note – Installation of this will in no ways affect the Setup of the Project and if you are using Cmder, run Cmder with Run as administrator. Tip – you can save a shortcut of this app to your desktop.
3. Your System should have installed Django (preferably the latest version). If you don't have Django installed then you can simply install it using pip as follows.

```
python -m pip install Django
```

4. Install Git to clone my GitHub repository to a desired folder of yours.

Setup –

1. First of all, clone my GitHub repository by HTTPS. You can do so by running the following command.

   git clone https://github.com/PRITISH009/Post-It.git

2. Now after you have successfully cloned the repository in your desired location, you need to install a few things to run the Application properly and to do so, run the following commands –

   python -m pip install django-crispy-forms

   python -m pip install Pillow

3. Now run server to start the application. To do so, go inside the PostIt Folder using Terminal and run the following command.

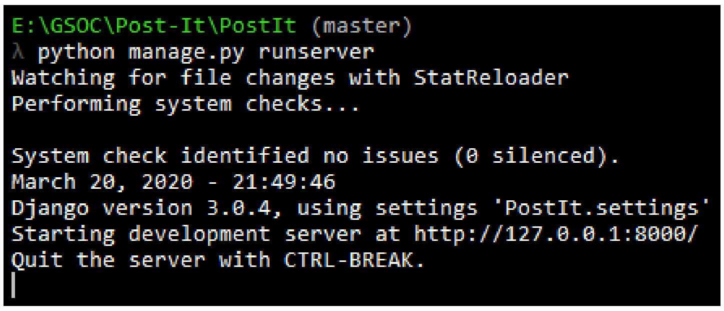
   python manage.py runserver

   Running this command starts the Django server on your localhost.

4. Now to view the app, go to the following URL –

   localhost:8000

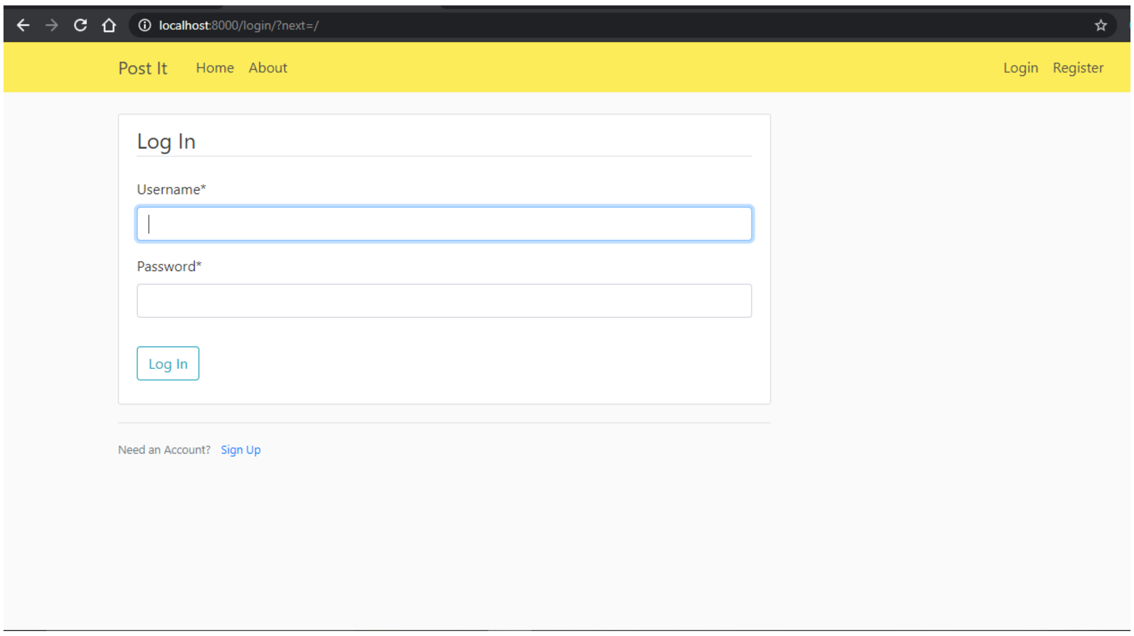
   If the app is successfully set up and the server is running properly then the following will be displayed in your terminal.



```
E:\GSOC\Post-It\PostIt (master)
λ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
March 20, 2020 - 21:49:46
Django version 3.0.4, using settings 'PostIt.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

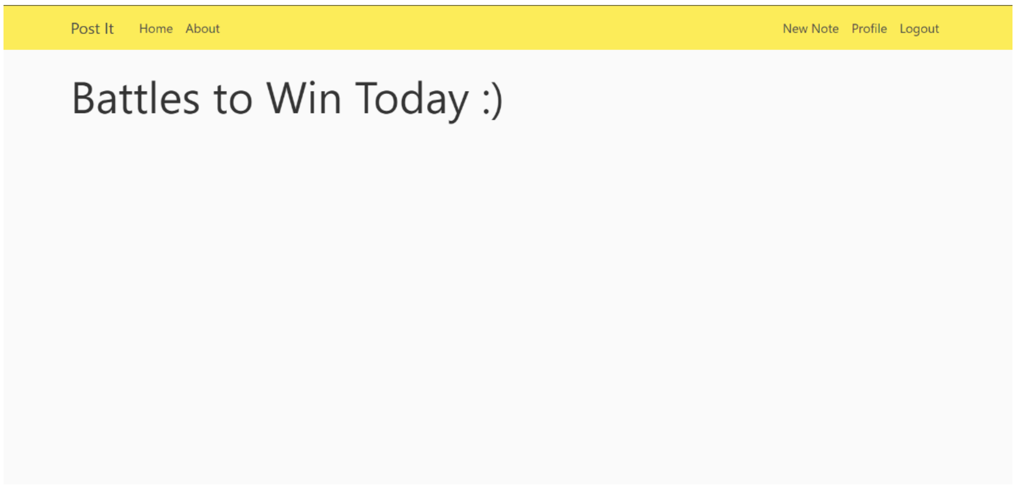
And on the web browser, it will show something like this



Here you can Log in or Sign Up using the given link below the form or the Register Link in the navigation bar.

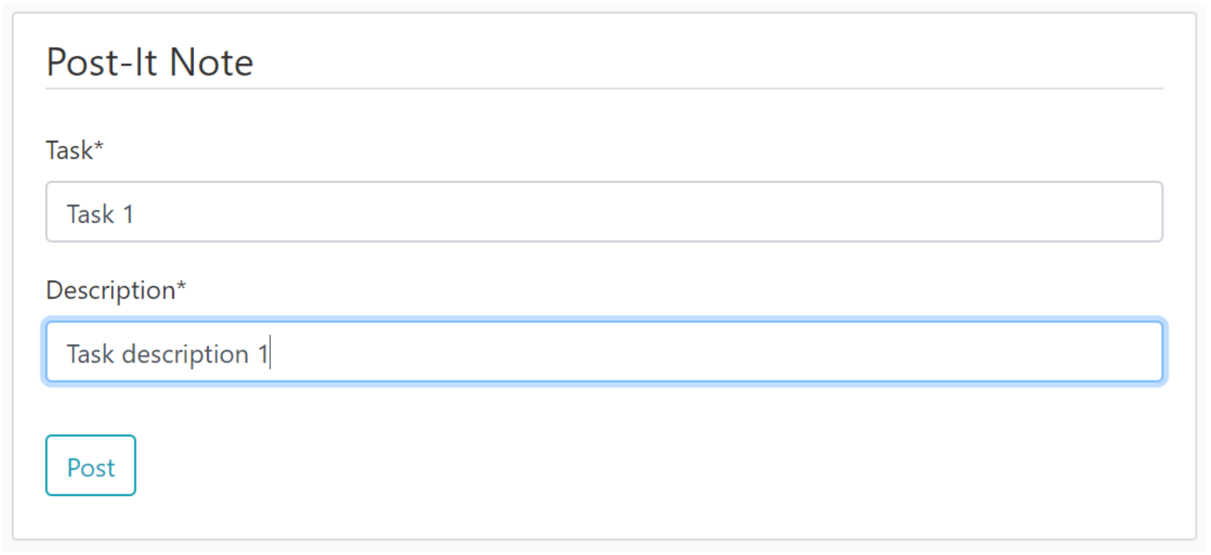
There are already 3 users created Admin, NewUser and Test. For Admin account the password is Admin and for the other two accounts the password is Testing321.

5. Now once you reach the home page (as shown below), you are all ready and set to manage all of your tasks. You can now create, update and delete your Post-It Notes here. (There are no Notes currently)



Functionalities Explained (Application Tutorial)

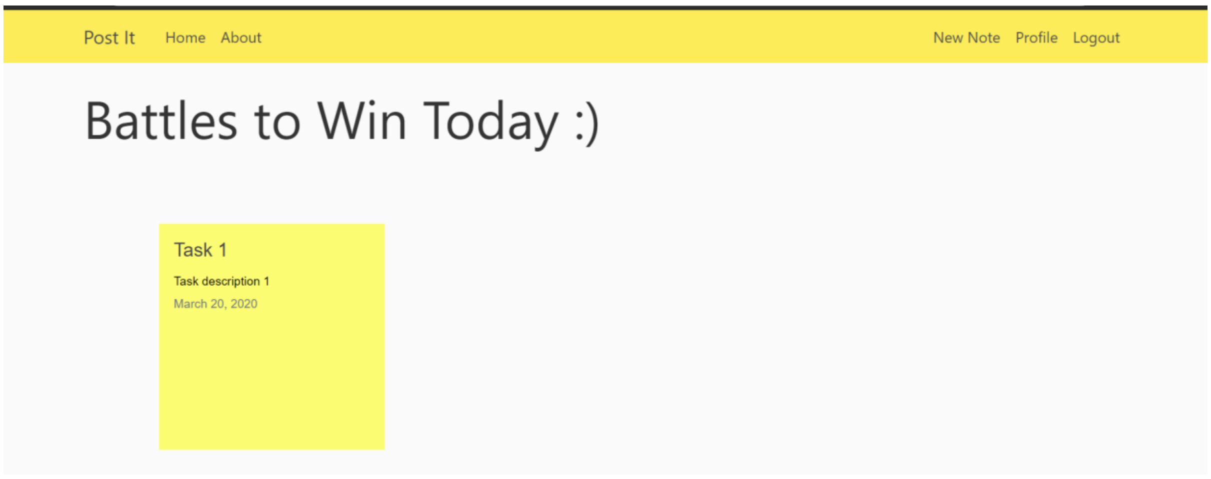
1. Creating a New Note – As you can see there is a link in the navigation bar in the Home Page which says “New Note”. You can click on this to Create a New Note and as soon as you submit you will be redirected to the Home page where your note will be displayed on the wall.



Post-It Note

Task*: Task 1

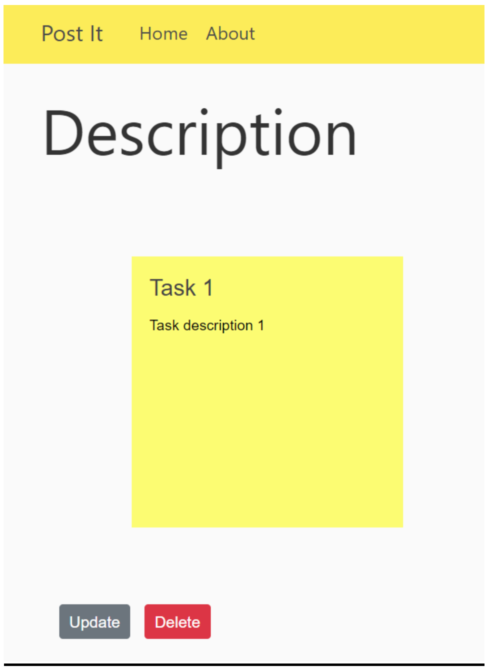
Description*: Task description 1

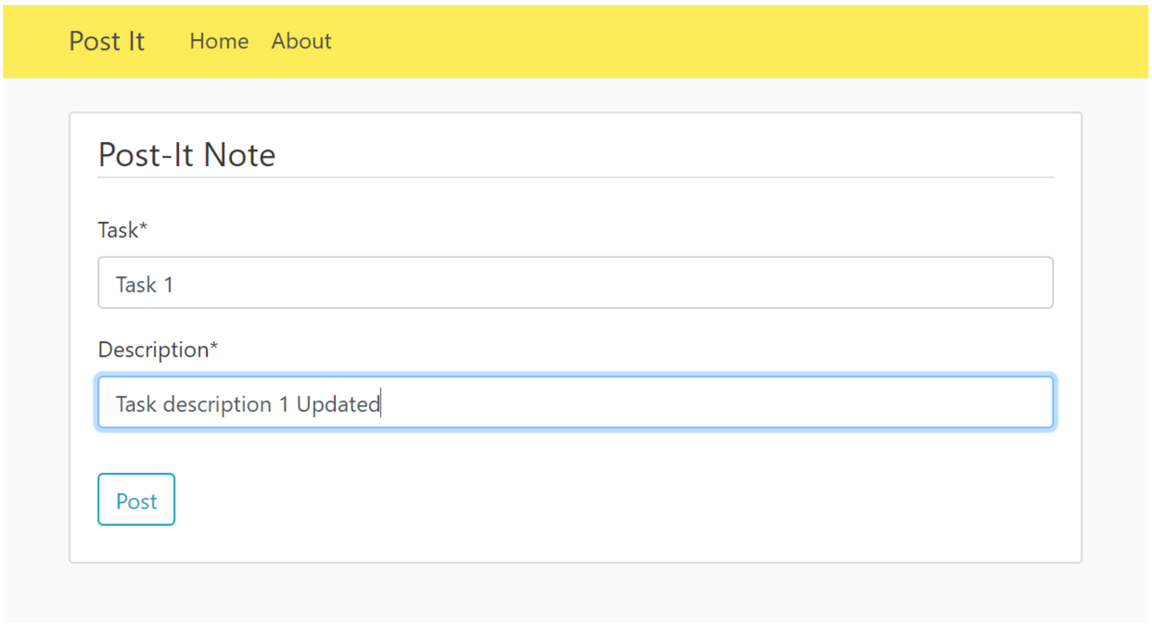


Battles to Win Today :)

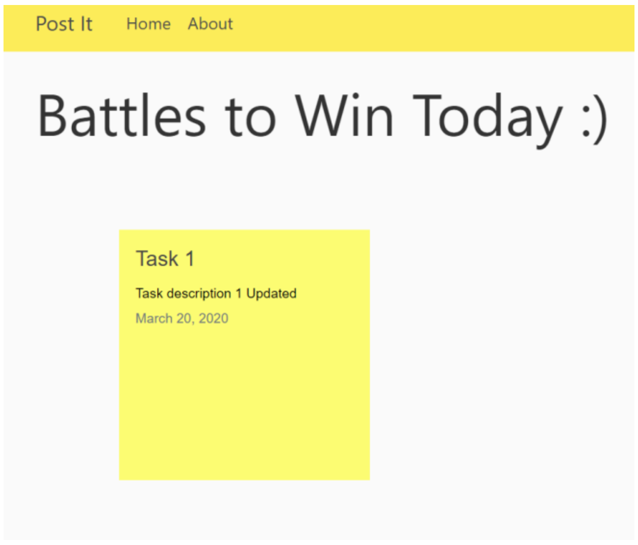
Task 1
Task description 1
March 20, 2020

2. Updating a Note - You can also update a note by just clicking on to your note. By doing so, the Details page of the Note will open up where you have the option to either update or delete the note.





Here you can change the details of the Post and Post again. After posting, the app will again take you to the Home page where you can see the Updated Post.



Post It Home About

Battles to Win Today :)

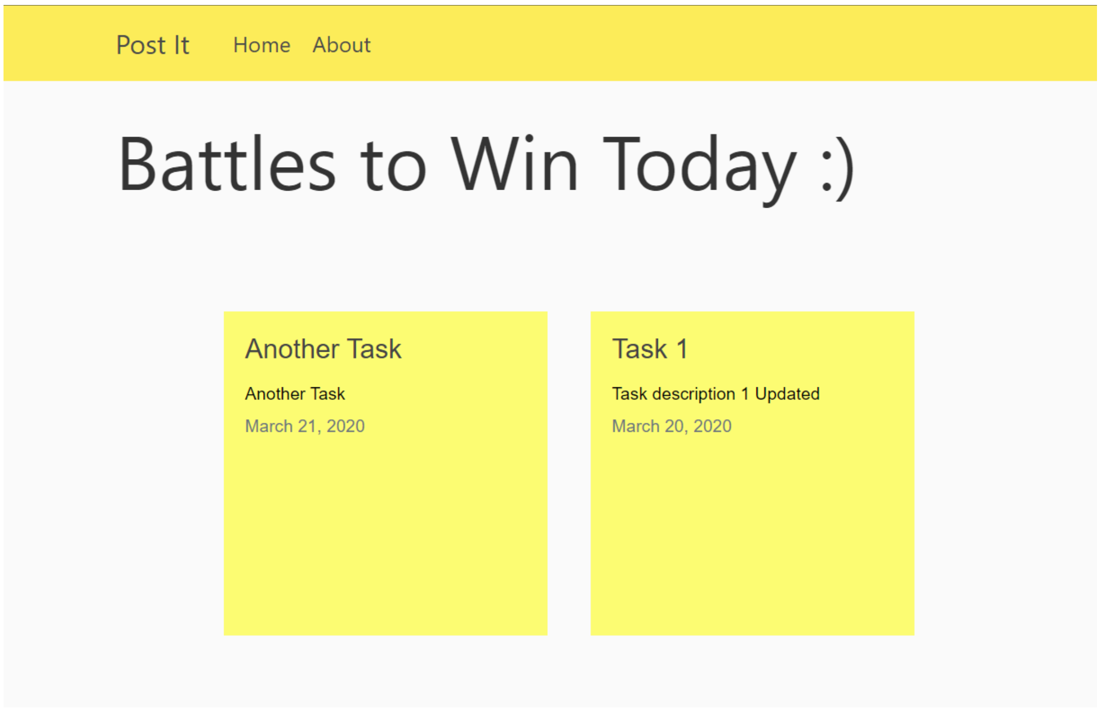
Task 1

Task description 1 Updated

March 20, 2020

3. Deleting a Note - To delete a note, the process is very similar to updating, instead of Update button just click on delete and confirm. The note will be deleted from the Wall.
4. Notes Sorted date wise - All the notes that are posted are displayed date wise which means the Note posted on the latest date will occur first and the ones that are older will come later.

   So, after adding another note the wall looks something like this -



Post It Home About

Battles to Win Today :)

Another Task

Another Task

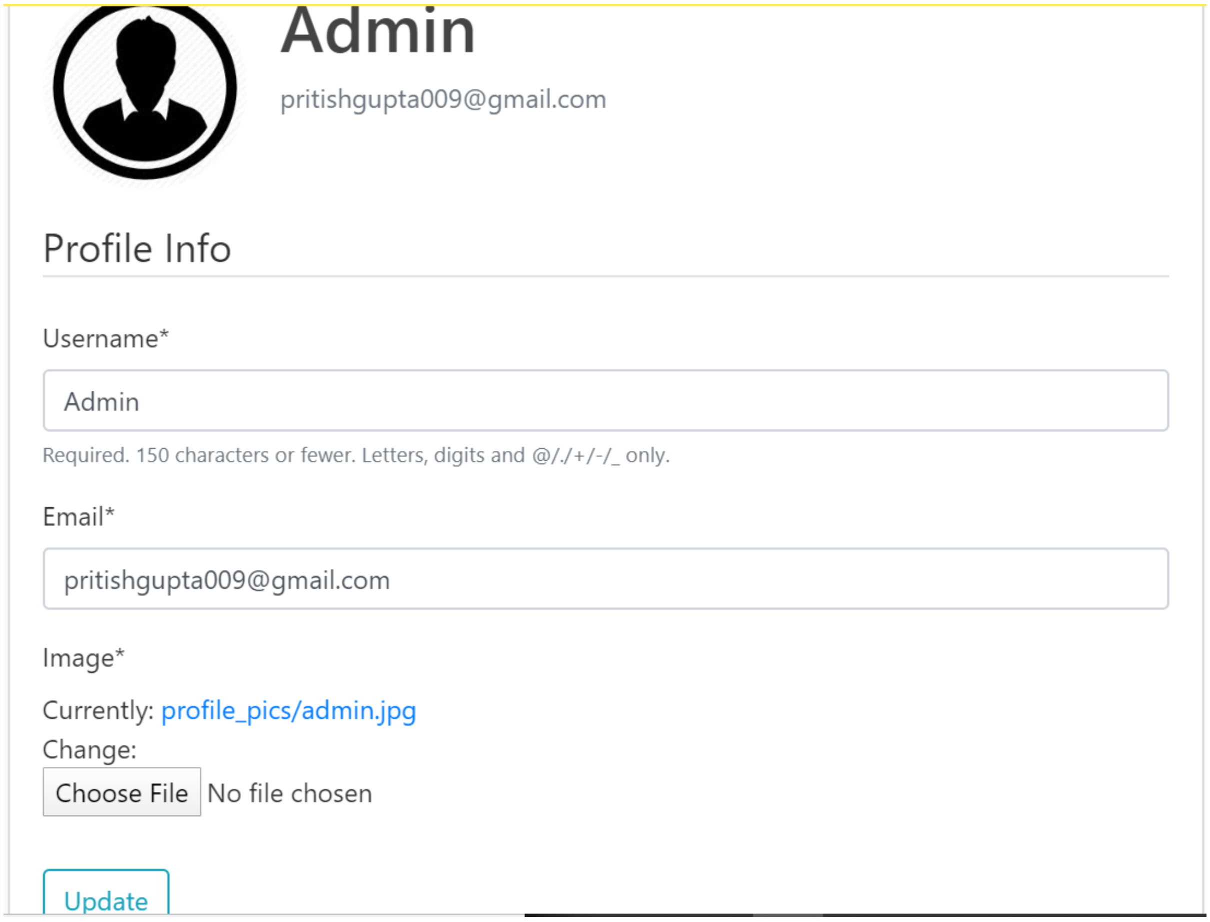
March 21, 2020

Task 1

Task description 1 Updated

March 20, 2020

5. Update User Profile - As each user has an account to himself/herself, each user is given the functionality to update his/her profile. There are a few different things that a user can change. For instance, a user can upload or update his/her Profile Pic, change email address linked to his/her account or can change his/her Username.


Admin

pritishgupta009@gmail.com

Profile Info

Username*

Admin

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Email*

pritishgupta009@gmail.com

Image*

Currently: profile_pics/admin.jpg

Change:

Choose File No file chosen

Update

6. Pagination – To avoid cluttering of all the notes at one place, Pagination has been used which allows the application to help create multiple pages where each page can have at maximum 4 Notes and all the other notes are shifted to the next page. The latest 4 Notes will be displayed first and the rest will be on the next pages.

Code Documentation

This section contains details about the code written in the GitHub Repository.

<u>**General Overview of the Project directory**</u>

The Application consists of the following sub directories –

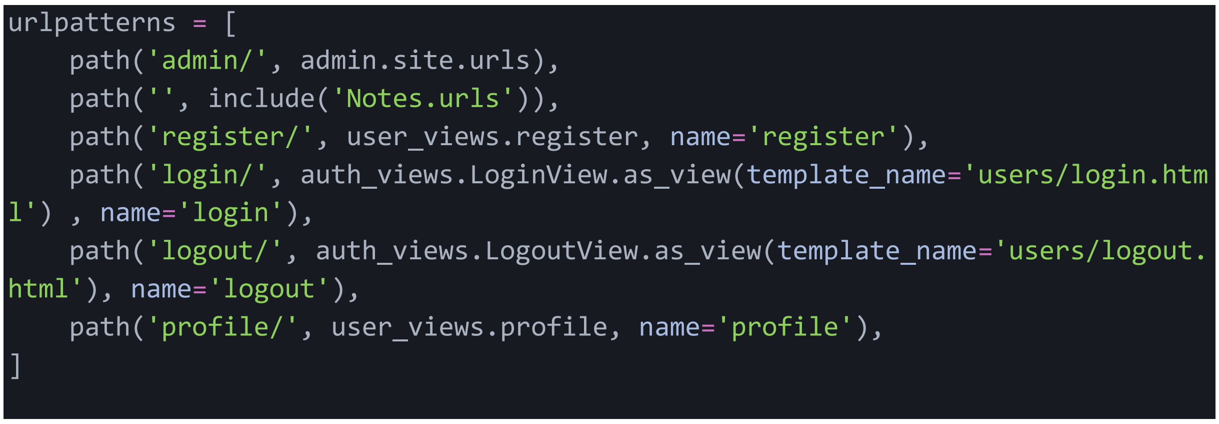
1. <u>PostIt</u> – This is the main project and consists of main urls.py where Django looks for its URLS.
2. <u>Notes</u> – This is a sub App installed in the project. This section deals with everything about Post It Notes from creation, updation and deletion.
3. <u>Users</u> – This is again a sub App which handles everything related to users from profile creation and updation.

Each of the sub app consists of a templates directory which consists of all the templates used to render the view on the WebApp. Each of these sub apps also consists of models.py file which consists of databases related to the respective sub app (e.g. Notes consists of a table which stores fields related to Notes table of the project).

Also, the Notes app consists of its own **urls.py** which handles routing related to the Notes app. Whereas everything related to routing in the users app is in the main project urls.py.

Routing in the Project

Starting off with main **urls.py.** This page consists of the following URLs –



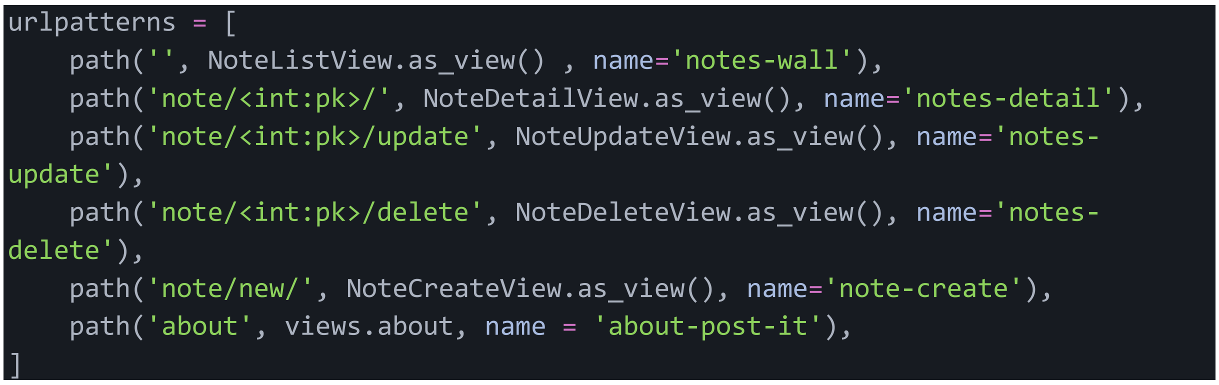
```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('Notes.urls')),
    path('register/', user_views.register, name='register'),
    path('login/', auth_views.LoginView.as_view(template_name='users/login.htm
l') , name='login'),
    path('logout/', auth_views.LogoutView.as_view(template_name='users/logout.
html'), name='logout'),
    path('profile/', user_views.profile, name='profile'),
]
```

- path('admin/' , admin.site.urls) – leads to the admin page of Django where only the Admin of the web-app can login. This page allows the admin to manually handle everything from updating User Profile to Notes etc.
- path(' ', include('Notes.urls')) – This tells Django to redirect to Notes app for the Home page of the WebApp. The home page of this app is nothing but the Wall where the user posts all of his/her Notes.
- path('register/', user_views.register, name='register') – This path leads the user to the users app's views.py file. In this file the register function is invoked (as specified in the path).
- path('login/', auth_views.LoginView.as_view(template_name='users/login.html') , name='login') – This path leads the user to the LoginView from predefined Class Based Views imported from django.contrib.auth module. This is passed a template from 'users/login.html' from the template module in the user's app.

- Similarly, for /login and /logout route. All of these routes call their respective Class Based Views and are pass their respective templates to render a web page.
- path(‘profile/’, user_views.profile, name=’profile’) – Similarly this route calls the profile function defined in the users views to generate the web interface for the respective task.

Similar to the main urls.py the Notes app also consists of it’s own urls.py for handling every functionality with the Post-It Notes from creation, deletion and updation.

The users/urls.py consists of the following routes –



```
urlpatterns = [
    path('', NoteListView.as_view() , name='notes-wall'),
    path('note/<int:pk>/', NoteDetailView.as_view(), name='notes-detail'),
    path('note/<int:pk>/update', NoteUpdateView.as_view(), name='notes-
update'),
    path('note/<int:pk>/delete', NoteDeleteView.as_view(), name='notes-
delete'),
    path('note/new/', NoteCreateView.as_view(), name='note-create'),
    path('about', views.about, name = 'about-post-it'),
]
```

As you can see, each of these routes calls different custom CBV (Class based view) for handling various functionalities. Each of these views are defined in the users/views.py.

Note – the ‘name’ field in the path is very important as it is a direct reference to each of these paths and are used as links in the templates to jump from one page to another.

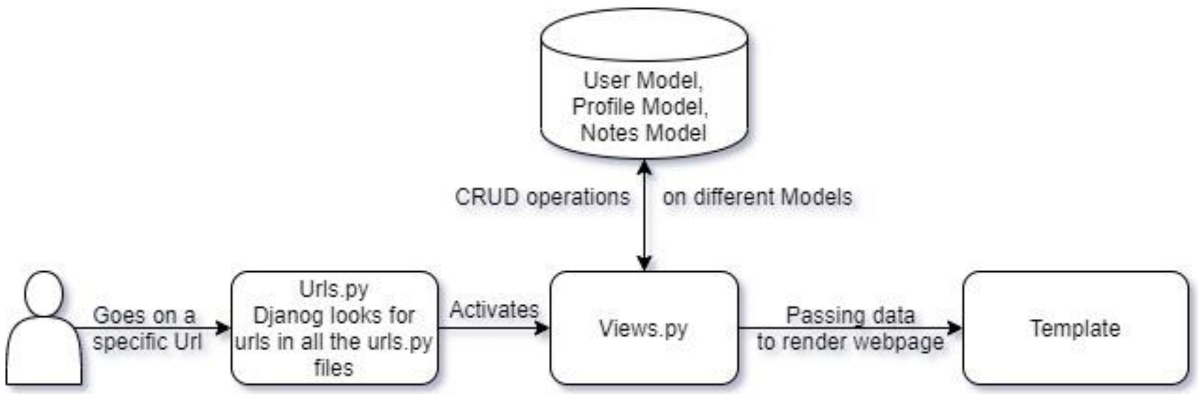
Working – So, whenever a user calls a specific URL, Django first looks into the main urls.py file and if a direct link is found in this file, then with the help of the second argument of that path(), triggers a specific function to handle the functionality along with passing in a template to render the web interface.

For example –

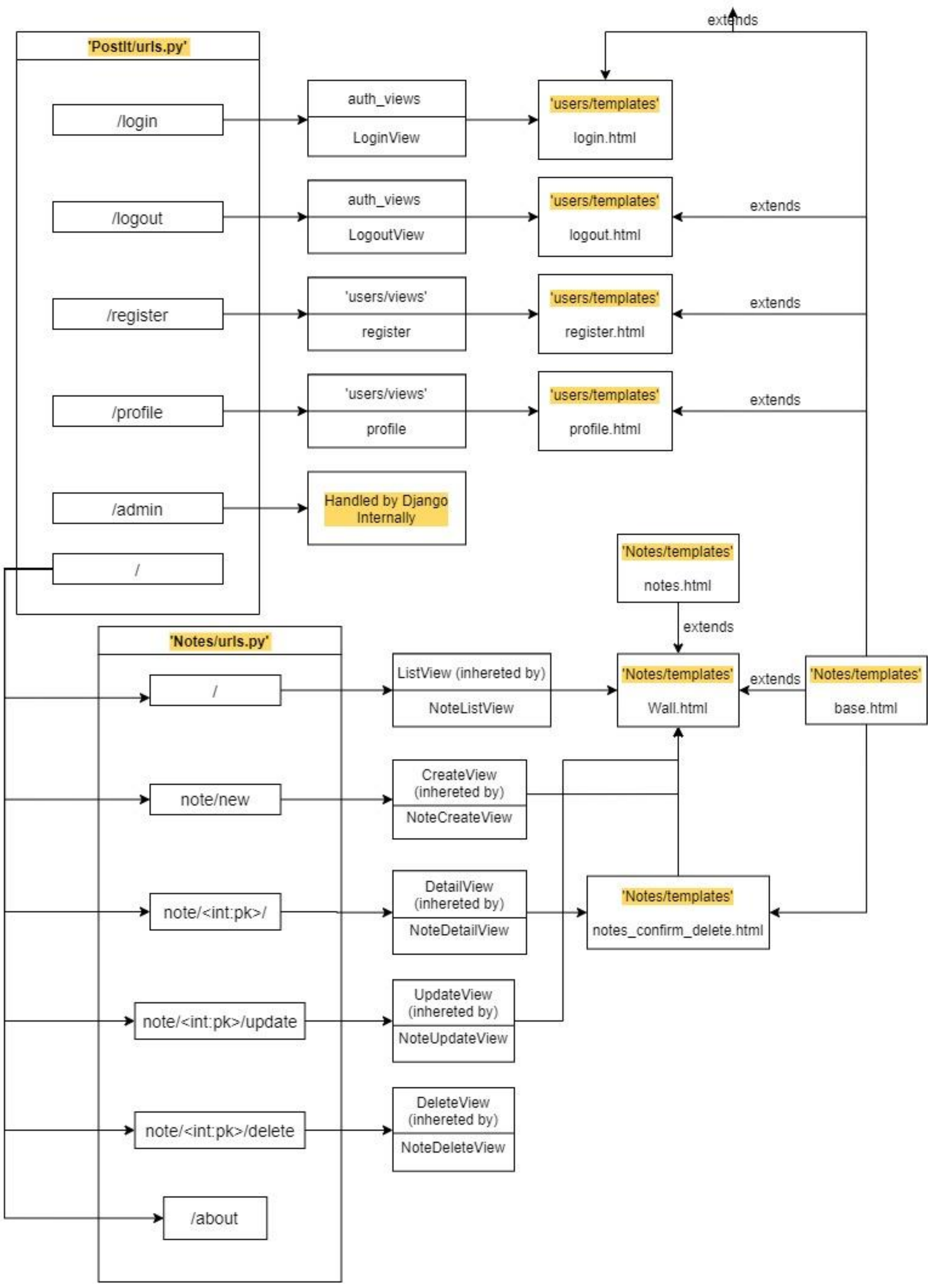
When a user goes to the home page of the WebApp, it basically means that the user is calling for ‘localhost:8000/’ URL. The localhost:8000 part is removed from the URL and then Django looks for this path in it’s main urls.py file. Here Django see’s that this path specifically asks Django to include URLs listed in the users/urls.py file. Here since there is nothing after the ‘/’, it looks for an empty URL i.e. ‘notes-wall’ path and renders the **NotesListView** (class-based view defined in the users/views.py).

Similarly, for /profile or /login routes the paths are already defined in the main urls.py file hence when one of these or similar routes are called then the respective View is automatically rendered. So, the /profile path will go on to display the profile page of the user whereas the /login passes a template from ‘users/login.html’ and displays a predefined django authentication form imported from django.contrib.auth.

Working Mechanism of the Project



System Workflow –



Web Interface through Class Based Views

Mostly all the pages in the project are displayed with the help of **Class Based Views.** Where these CBVs are either predefined in django itself or are custom made. Each of these views are passed an html template with some context thus rendering a web interface to the user.

The context here can be either a form for the purpose of handling a functionality such as Login or Register etc. thus giving user an interface to do the same or it can be some plain data either from the database (e.g. - all the notes objects created by user for his/her wall) or hardcoded (e.g. - about page).

This section explains different CBVs in detail –

There are two views.py file in the project, 1. In Notes App (to handle all the functionalities related to Notes) and 2. In users folder (to handle all the functionalities regarding User (Registration and Profile)

In the Notes/views.py there are several CBVs defined to handle different aspects of Post-It Notes. These CBVs are discussed as follows -

I. NotesListView

```
class NoteListView(LoginRequiredMixin,ListView):
    model = Notes
    context_object_name = 'posts'
    template_name = 'Notes/wall.html'
    paginate_by = 4

    def get_queryset(self):
        queryset = Notes.objects.filter(user = self.request.user.id).order_by(
'-date_posted')
        return queryset
```

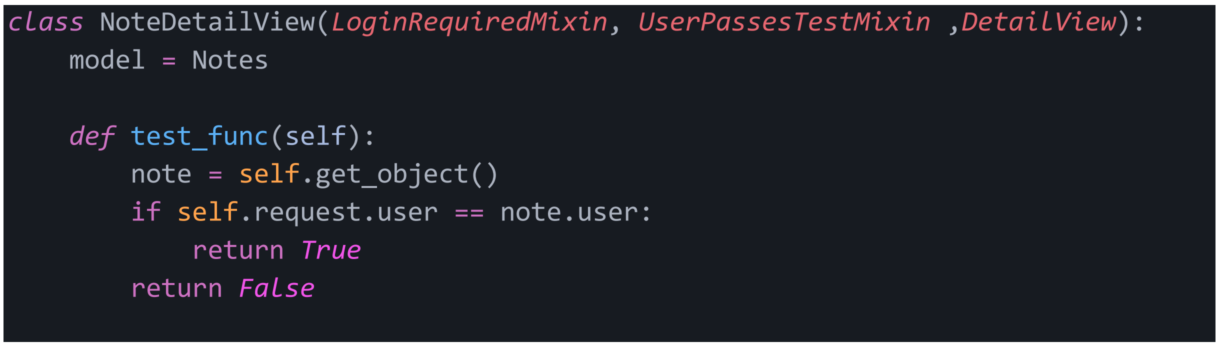
NotesListView is a custom CBV inherited from predefined ListView in django. Here it gets all the notes objects to display from the **get_queryset** function which returns an ordered set of notes from the Logged-In User. The view takes this object and names it with the value in **context_object_name** i.e. posts and passes it to the template 'Notes/wall.html'.

The CBV is also paginated with the help of the field **paginate_by** which is set to 4 which means that all the notes will be divided into groups of 4 per page rest of the notes will be displayed on the next page. The notes objects passed by the get_queryset function are also ordered by '-date_posted' which means that all the newest notes will be displayed first and the rest will be displayed later.

Note – The LoginRequiredMixin is used to authenticate the user. If the user is not logged in it will redirect the user to the Login page with the help of LOGIN_ _URL value stored in the settings.py file of the Application.

In some of the CBVs UserPassesTestMixin is also used which helps add another level of authentication so as to identify that a Logged In user is the correct user to perform a specific task like updation or deletion of a note (from url).

II. NotesDetailView



```
class NoteDetailView(LoginRequiredMixin, UserPassesTestMixin ,DetailView):
    model = Notes

    def test_func(self):
        note = self.get_object()
        if self.request.user == note.user:
            return True
        return False
```

This CBV is extended from the predefined DetailView of the django to handle the description page of each of the Notes. When a note from the Wall is clicked, a description page of the note is shown. This description page also consists of two functionalities from Update and Delete.

Here only the user who is the author of the note has the access to the description page. With the help of **test_func** the correct user is authenticated. If the user is not the author of the note and tries to access other's note then a **403 Forbidden** response is given to the invalid user.

III. NoteCreateView

```
class NoteCreateView(LoginRequiredMixin,CreateView):
    model = Notes
    fields = ['task', 'description']
    success_url = '/'
    def form_valid(self, form):
        form.instance.user = self.request.user
        return super().form_valid(form)
```

NoteCreateView extended from CreateView is used for handling Note Creation functionality of the Application. When a user clicks on **New Note** in the navigation bar, a NoteCreateView is rendered.

A predefined django form is passed to the NoteCreateView the web interface is rendered showing a form to add details about the Note namely task and description. The fields to be filled by user are defined in this CBV with the help of "fields" attribute. When this form is submitted, the form_valid function is triggered validating the essentials like if the fields are filled or not. The form validation is completely handled by this CBV.

IV. NoteUpdateView

```
class NoteUpdateView(LoginRequiredMixin, UserPassesTestMixin, UpdateView):
    model = Notes
    fields = ['task', 'description']
    success_url = '/'
    def form_valid(self, form):
        form.instance.user = self.request.user
        return super().form_valid(form)

    def test_func(self):
        note = self.get_object()
        if self.request.user == note.user:
            return True
        return False
```

This CBV (inherited from UpdateView) is used to render the web interface for Updating an already created Note. Here as you can see, the LoginRequiredMixin is used to check if the user is authenticated or not and UserPassesTestMixin is used to check if the author of the post is the only person who is Updating the Note.

Again, similar to the NoteCreateView a form is passed with fields task and description which holds the previous values and can be updated and submitted. Again, form_valid function is used to validate the correctness of the form and test_func function is used authorize the correct user to update the post.

V. NoteDeleteView

```
class NoteDeleteView(LoginRequiredMixin, UserPassesTestMixin, DeleteView):
    model = Notes
    success_url = '/'
    def test_func(self):
        note = self.get_object()
        if self.request.user == note.user:
            return True
        return False
```

This is again a view extended from DeleteView in django. It uses LoginRequiredMixin and UserPassesTestMixin to authenticate and authorize the correct user to delete a post. This deletes the note from the Notes model in the models.py.

Note – Here all the above CBVs discussed above use the Notes model (i.e. Notes table (explained later in detail)) defined in models.py file in the Notes app.

Note – For Login and Logout functionalities, Django's predefined views from auth_views are directly used hence are not defined anywhere.

The views discussed below are Function Based Views (FVB) in PostIt/view.py. The views here manage the user registration and profile creation.

VI. register FBV

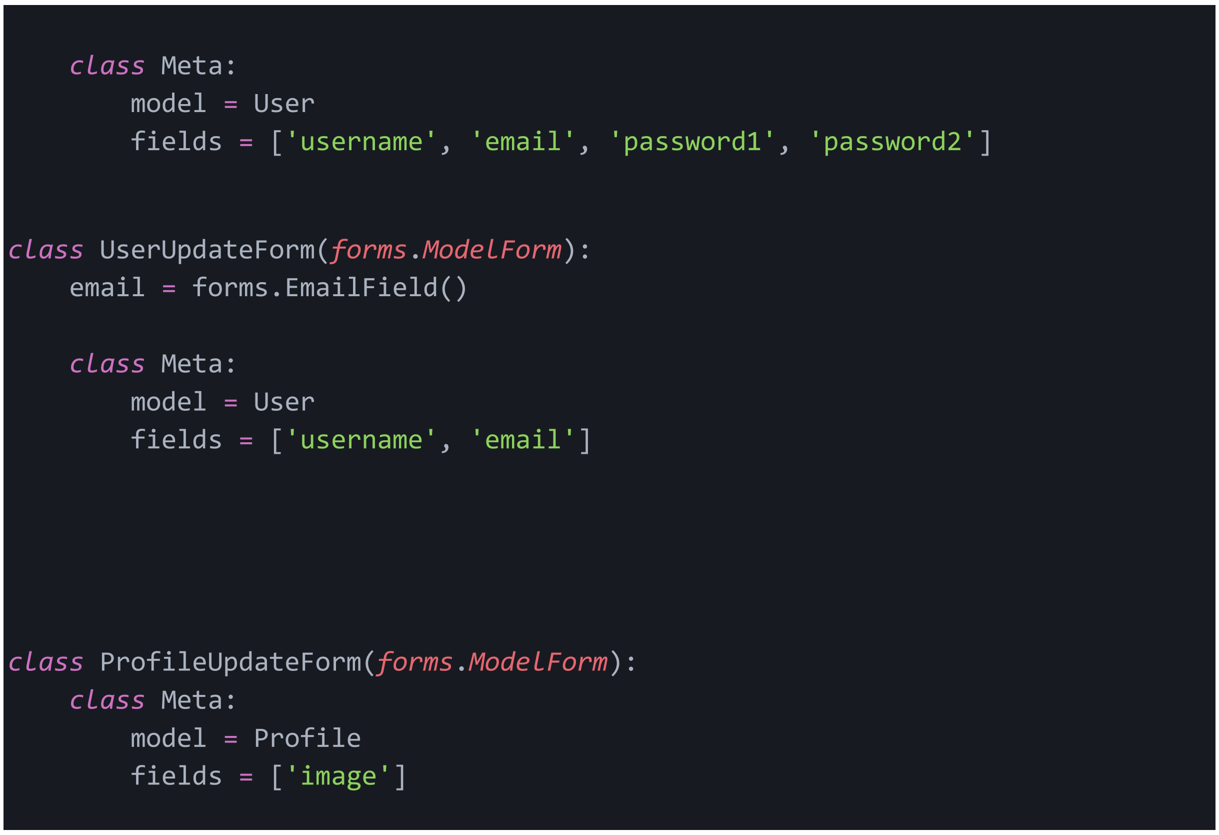
```
def register(req):
    if req.method == 'POST':
        form = UserRegisterForm(req.POST)

        if form.is_valid():
            form.save()
            username = form.cleaned_data.get('username')
            messages.success(req, f'Your Account has been Created! You can
now Log In')
            return redirect('login')


    else:
        form = UserRegisterForm()
    return render(req, 'users/register.html', {'form' : form})
```

register is a function-based view which uses a UserRegistrationForm from the forms.py file in the project. The forms.py file consists of the following forms –

```
class UserRegisterForm(UserCreationForm):
    email = forms.EmailField()
```



```
    class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']


class UserUpdateForm(forms.ModelForm):
    email = forms.EmailField()

    class Meta:
        model = User
        fields = ['username', 'email']





class ProfileUpdateForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['image']
```

All the above forms are either directly inherited (ProfileUpdateForm and UserUpdateForm) or modified a bit after inheriting from a different form (UserRegistrationForm).

How the register function works is as follows –

If a user clicks on the **Register** link in the navigation bar of the WebApp, a **GET** request is made to the register function. The function creates an empty UserRegistrationForm and passes it to the **'users/register.html'** template. All the form values are displayed.

Now when the user fills in all the details, validation of the form is automatically taken care off by Django and a **POST** request is made to the same register function but this time the details from the form is saved into the Users database.

VII. Profile FBV

```
@login_required
def profile(req):
    if req.method == 'POST':
        u_form = UserUpdateForm(req.POST, instance=req.user)
        p_form = ProfileUpdateForm(req.POST, req.FILES ,instance=req.user.p
rofile)

        if u_form.is_valid() and p_form.is_valid():
            u_form.save()
            p_form.save()
            messages.success(req, f'Your Account has been Updated!')
            return redirect('profile')

    else:
        u_form = UserUpdateForm(instance=req.user)
        p_form = ProfileUpdateForm(instance=req.user.profile)

    context = {
        'u_form' : u_form,
        'p_form' : p_form
    }

    return render(req, 'users/profile.html', context)
```

This method is to update a user profile for a logged in user hence a decorator "@login_required" is used in order to authorize and authenticate the correct user.

This is very similar to the above register function. The concept is the same, if a POST request is made to this function, then the user_form (containing the user details) (also an instance of UserUpdateForm from forms.py) and profile_form (containing the profile details like email username and Profile Pic) (also an instance of ProfileUpdateForm from forms.py) are saved to save the Profile details and User details in the database.

Whereas if a GET request is made then an empty form containing fields from both user_form and profile_form is displayed by sending these field values to the 'users/profile.html' template.

Models/Database Schema of the Project

There are two models defined in this project –

1. **Notes** model in 'Notes/models.py' and

2. **Profile** model in 'users/models.py'. (The project also contains Users table but that is internally handled by Django itself)

1. Notes Model –

```
class Notes(models.Model):
    task = models.CharField(max_length=1000)
    description = models.CharField(max_length=10000)
    date_posted = models.DateTimeField(default=timezone.now)
    user = models.ForeignKey(User, on_delete=models.CASCADE, blank=True, null=
True)

    def __str__(self):
        return self.task

    def get_absolute_url(self):
        return reverse('notes-detail', kwargs={'pk' : self.pk})
```

There are 4 fields defined in the model –

I. task – The heading of the task. This field allows character inputs to the task heading and offers a max length of 1000 chars.

II. description – The description of the task. This field also allows all characters and offers a max length of description as 10,000 characters.

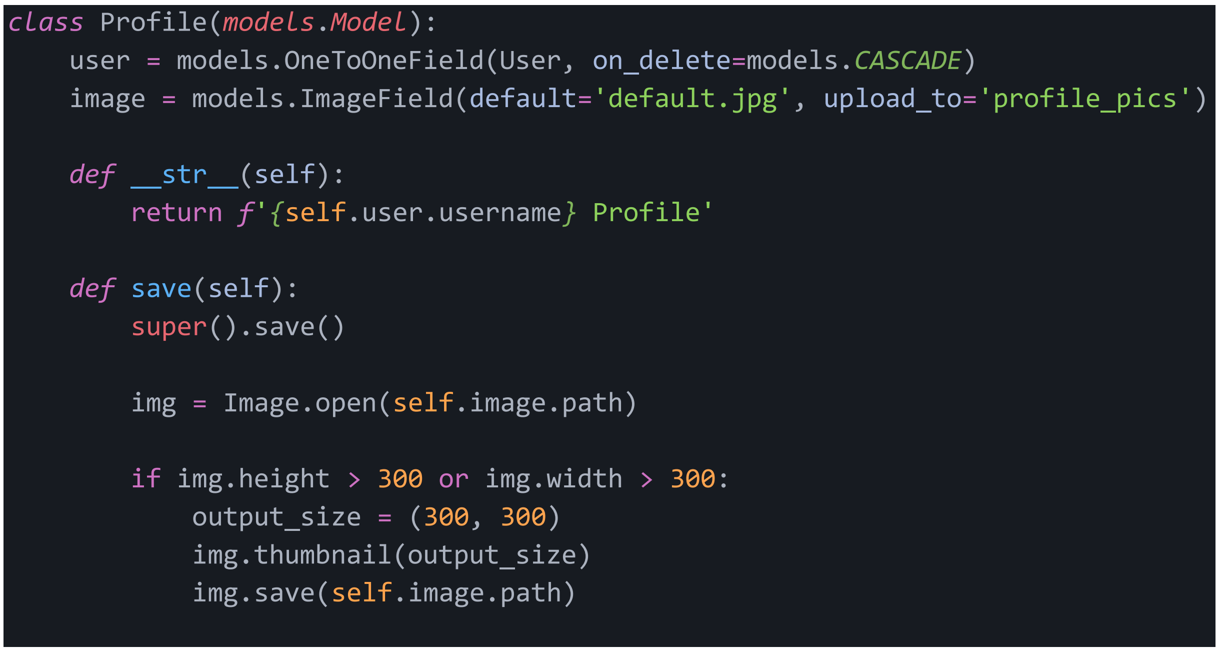
III. date_posted – The date and time of the note creation. This field automatically generates the Date and Time of the creation of the post.

IV. user – The user who wrote the note. This field is the foreign key from the internally defined and handled Users table from Django. On_delete = models.CASCADE is set so that if a user is deleted then all of it's notes are deleted but if a note is deleted User stays intact.

__str__ function is set so that as to define how will the results of a query be displayed when called from django shell.

get_absolute_url is used to provide the template of all Post-It Notes on the user's wall with a url which directs it to Note's description page.

2. Profile Model –



```
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    image = models.ImageField(default='default.jpg', upload_to='profile_pics')

    def __str__(self):
        return f'{self.user.username} Profile'

    def save(self):
        super().save()

        img = Image.open(self.image.path)

        if img.height > 300 or img.width > 300:
            output_size = (300, 300)
            img.thumbnail(output_size)
            img.save(self.image.path)
```

There are 2 fields defined in the model –

I. user – This field binds the profile of each user to a specific user defined in the inbuilt Users table of Django.
II. Image – This field stores the image of the user as Profile Pic.

There are 2 functions more –

__str__ function as described above provides a way to represent profiles. Here in this case it returns the results with "username Profile" as the identifier.

save function here is over-ridden so as to provide a functionality to resize the image in-case the image uploaded by the user is very big in size. If the size of the image (any height or width) is > 300 pixels then it is cut short and the resized image is saved. This reduces the load on backend when loading the profile page of the user and also saves a ton of space in backend.

One More thing about Saving Profile Pictures – All the photos are saved in the 'media/profile_pics/' directory of the project. A default picture is also kept in the media directory so that if a user doesn't upload a profile pic this default pic will be used to display his/her profile.

To do so the following lines are added to the settings.py file of the project.

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

Some other lines are also added to the settings.py to configure certain things. Such as –

To configure crispy forms to bootstrap4 the following line was added –

```
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

And for Redirection –

```
LOGIN_REDIRECT_URL = 'notes-wall'
LOGIN_URL = 'login'
```

When a user tries to access a page that requires the user to log in, using LOGIN_URL the user is redirected.

LOGIN_REDIRECT_URL is used to again redirect the user after his/her successful logging in to the page which the user was initially trying to access.

So Far –

So, we have looked at the different urls in the project and how these urls work. We have also looked at all the views and what each of the view is doing and what models are used in the project.

To put it briefly, all the Note based views (NoteCreateView, NoteDeleteView, NoteUpdateView, NoteDetailView and NoteListView) uses the Notes model in the project and deals with its objects to handle all the Notes related functionalities.

For instance – if '/note/new' is called then NoteCreateView is triggered which renders a predefined form. The details from this form are saved to the Notes database created with the help of Notes model.

register view (Function based View) uses the internally defined Users model from django to register a User through predefined forms.

And the profile view uses both Users (internally defined) and Profile model defined in the users/model.py to create and update a user’s profile through the UserUpdateForm and ProfileUpdateForm.

Templates –

Finally, all the data from the Views are passed on to the passed templates. All the templates are extended from a base template named **base.html** in the ‘Notes/templates/Notes/’ directory.

Apart from having hardcoded html tags it also contains some logic providing the application with some basic yet sensible functionalities. These are explained in detail below.

For example, look at the following code snipped from **base.html** –

```
<div class="navbar-nav">
                {% if user.is_authenticated %}
                <a class="nav-item nav-link" href="{% url 'note-
create' %}">New Note</a>
                <a class="nav-item nav-
link" href="{% url 'profile' %}">Profile</a>
                <a class="nav-item nav-
link" href="{% url 'logout' %}">Logout</a>
                {% else %}
                  <a class="nav-item nav-
link" href="{% url 'login' %}">Login</a>
                  <a class="nav-item nav-
link" href="{% url 'register' %}">Register</a>
                {% endif %}
```

Here using Jinja templates, an **if block** is created which checks, **if a user is authenticated then show the user Profile and Logout Options else show the user Login and Register Options**.

It makes complete sense as you don't want the logged-out user to see a Profile link or Logout link.

There are 2 templates directories in the project –

1. Notes/templates – This contains all the templates related to Notes and about.
2. users/templates – This contains all the templates regarding login, logout or register.

The base.html file contains a code block where each of the extended templates add their own content. This content is created from the context passed on to the template from different Views.

```
{% block content %}{% endblock %}
```

A few of the templates are discussed below to show how the templates work. The rest of the templates are very similar to the discussed templated below hence are not discussed. But for any of the not discussed templates you can easily refer the below templates.

I. **about.html** –

```
{% extends "Notes/base.html" %}
{% block content %}
    <p>This is a Web App that you can use to Post all your Notes, Tasks etc
. to boost your Productivity </p>
{% endblock content %}
```

For instance, here the block content is added with a paragraph tag containing the details about the application.

II. **wall.html** –

This file contains 2 blocks of code –

a)

```
{% for note in posts %}
   {%include "Notes/notes.html" with note=note %}
{% endfor %}
```

This executes a for loop to add another part of template to this template. The template added is notes.html (discussed next).

b)

```
{% if is_paginated %}

    {% if page_obj.has_previous %}
        <a href="?page=1" class="btn btn-outline-info mb-4">First</a>
        <a href="?page={{ page.obj.previous_page_number }}" class="btn btn-
outline-info mb-4">Previous</a>
    {% endif %}

    {% for num in page_obj.paginator.page_range %}
        {% if page_obj.number == num %}
            <a href="?page={{ num }}" class="btn btn-info mb-4">{{ num }}</a>
        {% elif num > page_obj.number|add:'-
3' and num < page_obj.number|add:'3' %}
            <a href="?page={{ num }}" class="btn btn-outline-info mb-
4">{{ num }}</a>
        {% endif %}
    {% endfor %}

    {% if page_obj.has_next %}
        <a href="?page={{ page_obj.next_page_number }}" class="btn btn-
outline-info mb-4">Next</a>
        <a href="?page={{ page.obj.paginator.num_pages }}" class="btn btn-
outline-info mb-4">Last</a>
    {% endif %}
```

This here adds pagination to the page. The above code checks if the page is paginated only then the code block will be executed. The first if block in the code checks if the page object has a previous page. Only then it displays “First” and “Previous” Options in the page.

These “First” and “Previous” are the links to the first and the previous page of the current page object.

The next block helps provides link from to 2 pages before and 2 pages after the current page.

And finally, the last if block checks if there exists a next page if it does then links to both “Next” and “Last” page are provided.

III. notes.html

```
{% load static %}
<li>
    <a href="{% url 'notes-detail' note.id %}">
        <h5>{{ note.task }}</h5>
        <div>
            <small>{{ note.description }}</small>
        </div>
        <div>
            <small class="text-
muted">{{ note.date_posted | date:"F d, Y" }}</small>
        </div>
    </a>
</li>
```

This is the template that is included in the wall.html page to display each Note on the wall. This consists of note task and description with date on which the note was posted.

Each template is assigned different classes from the main.css in ‘Notes/static’ file.

IV. notes_form.html

```
{% extends "Notes/base.html" %}
{% load crispy_forms_tags %}
{% block content %}
    <div class='content-section'>
        <form method="POST">
```

```
            {% csrf_token %}
            <fieldset class="form-group">
                <legend class="border-bottom mb-4">Post-It Note</legend>
                {{ form|crispy }}
            </fieldset>
            <div class="form-group">
                <button class="btn btn-outline-
info" type="submit">Post</button>
            </div>
        </form>
    </div>
{% endblock content %}
```

This template is used when “New Note” is clicked. Here crispy forms tags are being to display the form fields (of the form created by django) properly.

So, when the form is posted a csrf token is taken (to avoid cross-site referencing) and the display of form is handled by crispy forms and the form is ready to be submitted.