# Workshop: Petstagram

This document contains the second part of the Petstagram Workshop. Today, we will create the **models** for the project. Then, we will connect **PostgreSQL** and migrate them. After that, we will work with the **Django admin site** to make CRUD operations with the models. And finally, we will **read** (select and filter) them **using python code**, and we will **present** the information for each model on the **"details" web pages**.

**Note: we will NOT work with the profile/ user model in the Python Web Basics Course.**

The full project description of the project can be found in the [**Workshop Description Document**](http://svn.softuni.org/admin/svn/python-web/Sept-2022/Python-Web-Basics/06-Workshop-Part-1/06-Workshop-Description.docx).

## Workshop - Part 2.1

### Creating the Pet Model

Let us start by **creating the Pet model**.

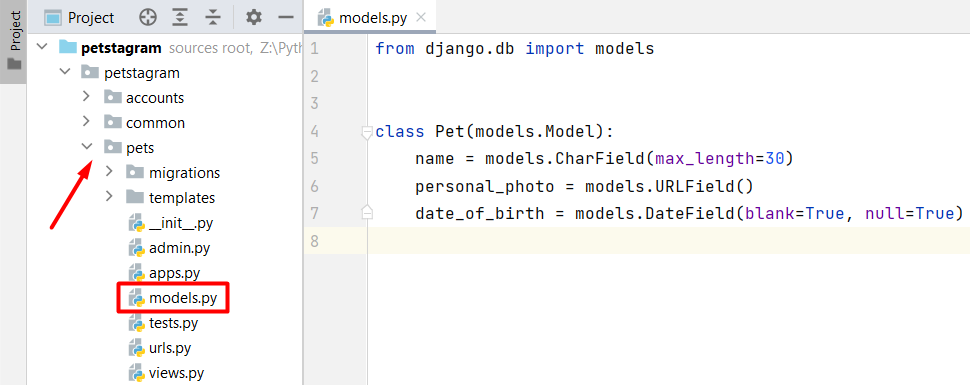
The fields **Name** and **Pet Photo** are **required**:

* **Name** - it should consist of a **maximum of 30 characters**.
* **Personal Pet Photo** - the user can **link a picture** using a URL

The field **date of birth** is **optional**:

* **Date of Birth** - pet's day, month, and year of birth

Open the **pets/models.py** file and let us create the model:



There should be created **one more field** that will be **auto-populated** with the following information:

* **Slug** - a slug automatically generated using the **pet's name and the pet's id, separated by a "-"** (dash).

The **slug is part of the URL** and as you know **each URL should be unique**: Graphical user interface, text, application

Description automatically generated

### Setting up the Database

Up to this moment, our Pet model is created and now we need to **migrate it to the database**. First, **start** the **PostgreSQL** container and the **PgAdmin** container (or you can create new ones): A screenshot of a computer

Description automatically generated with medium confidence

Wait a few seconds and **open the pgAdmin using the browser**: Graphical user interface, application

Description automatically generated

Then, log in with the **email** and **password** **you configure** (when creating the pgAdmin container):

Graphical user interface, website

Description automatically generated

You can **add a new server** to work with **or** **use the created one**:Graphical user interface, text, application, email

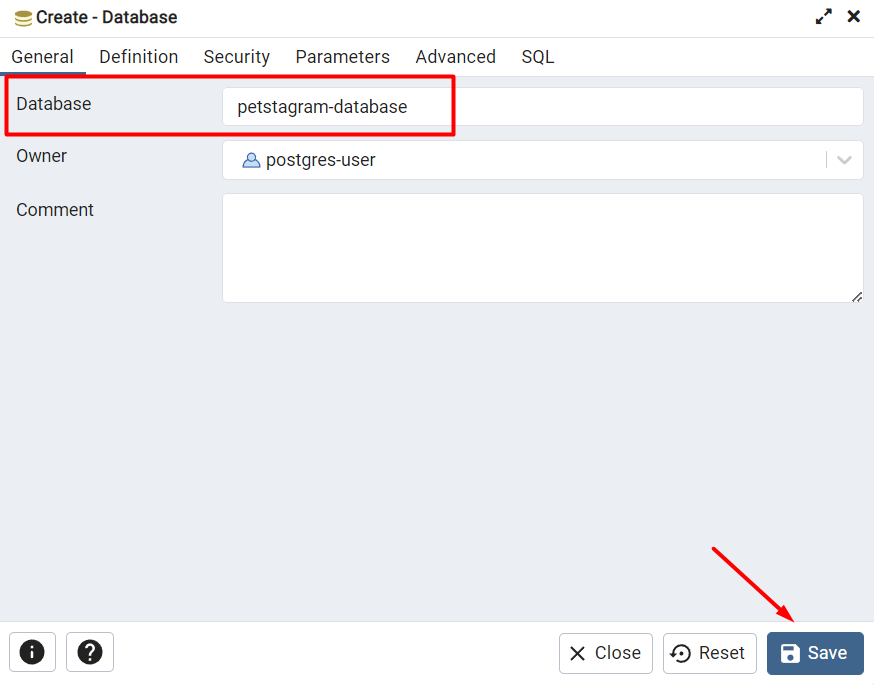
Description automatically generated

Next, on the server, we will **create the database we will work with**. Right-click on the "Database" field and choose to create a database:

Graphical user interface, text, application

Description automatically generated

Let us use the name "**petstagram-database**" for the project and **save it**:



Now, it is time to **configure it in the project**. Let us open the **settings.py** file and find the **DATABASES** setting:

Graphical user interface, text, application

Description automatically generated

Up until now, the project uses the default engine - SQLite. It is time to **write the configuration for the PostgreSQL**:

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.postgresql',  
 'NAME': 'petstagram-database', *# database name* 'USER': 'postgres-user', *# postgres user* 'PASSWORD': 'password', *# postgres password* 'HOST': '127.0.0.1', *# postgres host* 'PORT': '5432', *# postgres port* }  
}

The next required step is to **install psycopg2**. Open the Terminal and write the command **"pip install psycopg2**

**(python packages) "**:

Graphical user interface, text, application, email

Description automatically generated

### Migrate the Pet Model

When the installation is done, we can now **make the migration files** with the command "**python manage.py makemigrations**" and check if the migration file is successfully created:

Graphical user interface, text, application

Description automatically generated

Then, we can **migrate the changes** to the database using the command "**python manage.py migrate**":

Text

Description automatically generated

We can see that **not only our model was migrated** but some additional models are prebuilt in Django. Let us **check if our database is updated**. Follow the path **petstagram-database 🡪 Schemas 🡪 public 🡪 Tables**:

Graphical user interface, application

Description automatically generated with medium confidence

When we open the created table **pets\_pet** from our **Pet** model, **we can see all columns** we added when defining it. (Note: to see the table right-click on the **pets\_pet** and choose "View/Edit Data" -> "All Rows"):Graphical user interface, text, application

Description automatically generated

### Work with the Django Admin Site

Let us now work more with the model in **the Django admin interface**. First, open the **pets/admin.py file** and register the model on the admin site: Graphical user interface, application

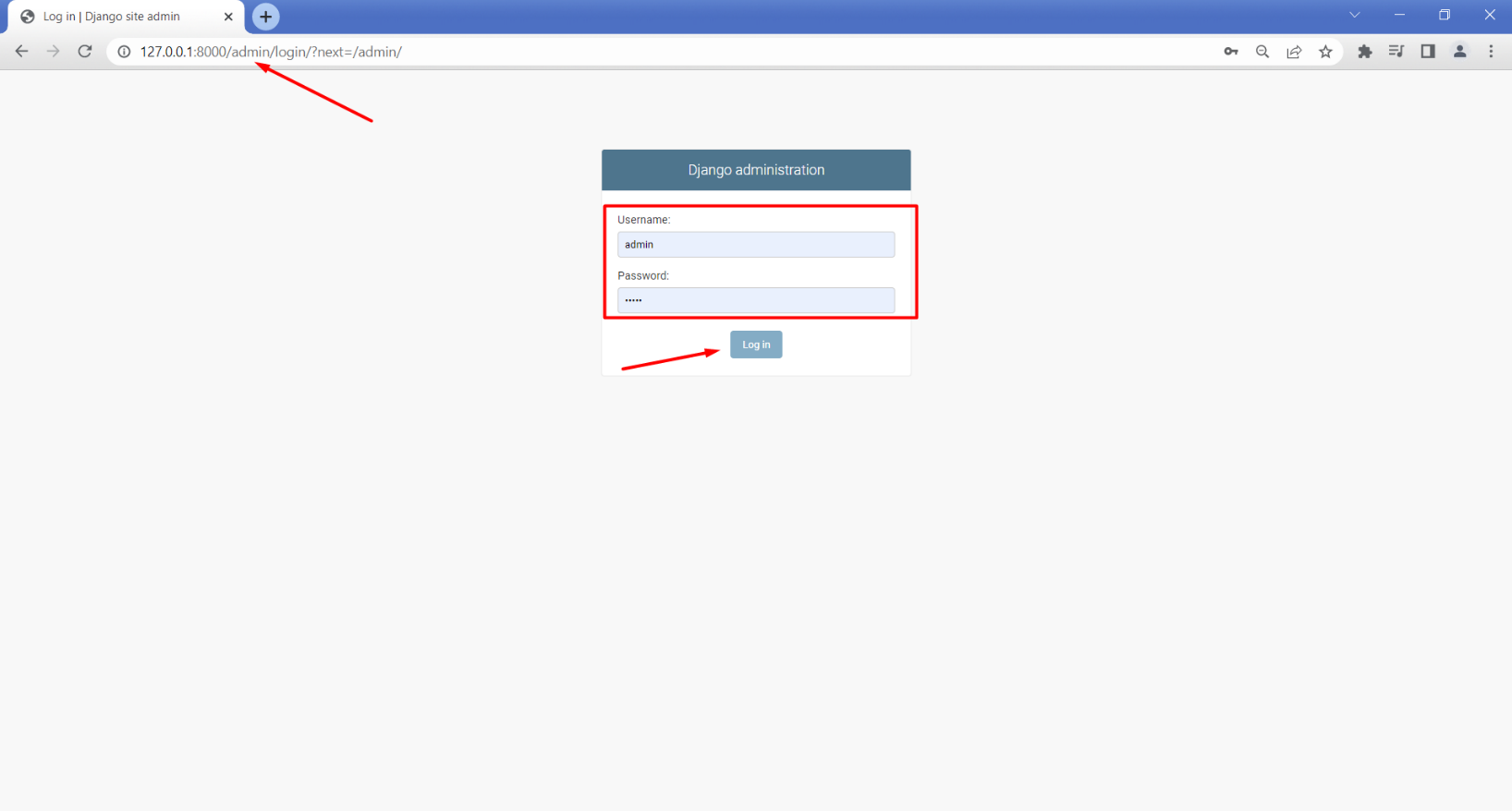
Description automatically generated

Next, to be able to **login to the admin site** (accessible only by admins) we must register as an administrator - it means **manage.py createsuperuser**":Graphical user interface, text

Description automatically generated

Right after we execute the command, Django asks us to **create a username** (in this case the username is "admin"), an **email address** (we can leave it blank just by clicking Enter), and a **password** (in this case the password is "admin"). (**Note**: In our case, Django asks us if we are sure we want to create an admin profile with a non-secure password. Let us type down "**y**" (for yes) as this is a personal project.)

Now, **start the development server**, go to the admin site at <http://127.0.0.1:8000/admin/>, and **log in with the credentials**:



As we log in, we could see all registered models. The one we need is Pets. Let us **add a new pet**: Graphical user interface, application, website

Description automatically generated

We can create a pet successfully. However, the **slug field is not auto-populated**:

Graphical user interface

Description automatically generated

To do that, we will **override the Pet model** **save()** **method** using a special function called **slugify()** which helps us structure a slug from a given value. The if-statement stands to say that the **slug field will NOT be changed when the name of the pet is changed**: Graphical user interface, application

Description automatically generated

We can go further and **change the slug field to be non-editable**. This way we ensure that the field will NOT be changed either in the form or the admin site:

Text

Description automatically generated

Now, we can improve the admin site interface by **visualizing the pet models in a human-readable way**:Graphical user interface, website

Description automatically generated

Up until now, each pet looks like a "**Pet object**" with an **id**. A thing we could do to ease the work of the administrators of the app is to **show each pet by its name**. Let us open the **pets/admin.py** file again and add a **list\_display** option: 

The admin interface changed like this: Graphical user interface, website

Description automatically generated

### Creating the Photo Model

It is time to create the second model for the pet's photo.

The field **Photo** is **required**:

* **Photo** - the user can **upload** a picture **from storage**, the **maximum** **size** of the photo can be **5MB**

The fields **description and tagged pets** are **optional**:

* **Description** - a user can write any description of the photo; it should consist of a **maximum of 300 characters** and a **minimum of 10 characters**
* **Location** - it should consist of a **maximum of 30 characters**
* **Tagged Pets** - the user can tag **none, one, or many** **of** **all pets**. There is **no limit** on the number of tagged pets

There should be created **one more field** that will be **automatically generated**:

* **Date of publication** - when a picture is **added** **or edited**, the date of publication is **automatically generated**

Open the **photos/models.py** file and let us create the model: Text

Description automatically generatedTo **work with an ImageField**, we should **install a library called Pillow**:

Graphical user interface, text, application

Description automatically generated

Note that the **photo field** has **additional validation for a maximum size of 5MB**. We should **create a** **custom validator** to validate the requirement. Let us create a new **validators.py** file in the **photos** app: Graphical user interface, application, Word

Description automatically generated

Open the file and **write the validation function** that will check if the photo size is above 5MB. In this case, it will raise a **ValidationError**: Graphical user interface, text, application

Description automatically generated

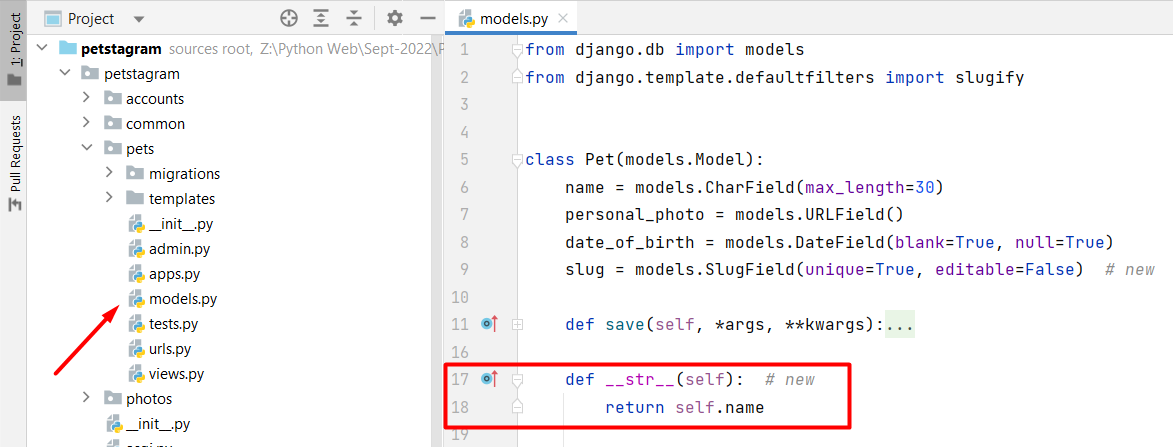
Then, we will **add the validator** to our **validators** list in the **photo** field:



Make **migration files** and **migrate the changes** to the database. **Register the model in the admin**.

Then, **start the development server** and check if everything **works correctly** in the admin panel: Graphical user interface, website

Description automatically generated

We can see that the pets in the tagged pets section **visualize as pet objects with an id**. We can change this by **overriding the \_\_str\_\_ method in the Pet model**: 

When we **reload** **the add photo page in the admin site** we will see the difference: Graphical user interface, website

Description automatically generated

Now, it is time to **customize the admin site interface** on the photos model page. Let us add a list of fields to be displayed on the photo model's page. The fields are the **id of the photo**, **date of publication**, **description**, and **names of all tagged** **pets**. We **cannot list a Many-to-Many field**, but we can **list the result of a function** that gets all objects from a Many-to-Many field and concatenate their names in a string: Graphical user interface, text

Description automatically generated

Now the **interface looks like that**: Graphical user interface, website

Description automatically generated

### Creating the Comment Model

It is time to create the comment model.

The field **Comment Text** is **required**:

* **Comment** **Text** - it should consist of a **maximum of 300 characters**

An additional field should be created:

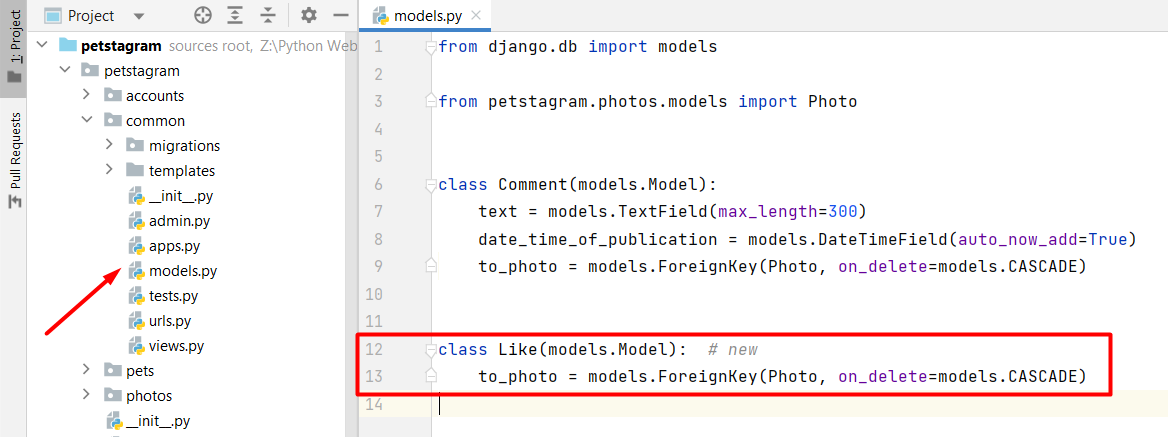
* **Date and Time of Publication** - when a comment is created (only), the date of publication is **automatically** **generated**

One more thing we should keep in mind is that **the comment should relate to the photo** (as in social apps users comment on a specific photo/post, i.e., the comment object is always connected to the photo object).

Open the **common/models.py file** and let us **create the model**: Graphical user interface, text, application

Description automatically generated

### Creating the Like Model

Finally, create the **Like model** which should connect one photo to one user. However, **we do not have a user object**, so we will **just create the model and add the photo relation**:

**Make the migration files** and **migrate the changes to the database**. We can now **register** **the models in the Django** **admin** site and **check if they work correctly**.

## Workshop - Part 2.2

### Add models to Home Page

We are ready to add some functionality to our **Home** **page**.

The Home page consists of **pet posts.** First, we will configure:

* The **location** (if one is added)
* The **tagged** **pets** (if any are added) - if there is **more than one pet** tagged, they must be shown on **different lines**
* The **link to the photo details** page
* **Date** of publication or edition of the photo

Let us open the **common/views.py** file. We will **read all photo objects from the database** and **add them to a context** dictionary:

Graphical user interface, text, application, email

Description automatically generated

Now, we can **inject the information into the pets-posts.html template**. (Note: we will **use** **the string "username" in the pet details URL** to bypass the user implementation):

{% load static %}  
**{% for photo in all\_photos %}**   
*...*

*<!-- Start User Details and Image Location -->*

*...*  
 *<!-- if the photo has location -->* **{% if photo.location %}  
 <span>{{ photo.location }}</span>  
 {% endif %}**  
*...*  
 *<!-- End User Details and Image Location -->  
...*  
 *<!-- Start Tagged Pets -->***{% for pet in photo.tagged\_pets.all %}** *<!-- Link to First Tagged Pet Details Page-->* **<a href="{% url 'pet-details' "username" pet.slug %}">**  
 <p class="message">  
 **<b>{{ pet.name }}</b>**  
 </p>  
 </a>  
 **{% endfor %}**  
 *<!-- End Tagged Pets -->  
  
 <!-- Link to Photo Details Page -->* **<a href="{% url 'photo-details' photo.pk %}">**  
 <h4 class="details">See details</h4>  
 </a>  
  
 *<!-- Date of Publication -->* **<h5 class="postTime">{{ photo.date\_of\_publication }}</h5>**  
*...*  
**{% endfor %}**

### Implement Like Button Functionality

Next, we will **implement the like button** and the **number of likes per photo**.

Let us start by creating a like button functionality - to work with the like button we should **create a view** **with the specific functionality**. First, create a **like button path** in the **common/urls.py urlspatterns** list: Graphical user interface, application, Word

Description automatically generated

Now, **create a** **like\_functionality** **view** in the **common/views.py**. The view will **receive the id of the current photo** and will **get the photo by the given id**. Then, the **view tries to filter the Like objects by the photo id** - if it finds an object, it means that the photo is liked. Based on that**, if the object is liked the view will delete the like** (and the object will be unliked). Otherwise, **the view will create a new Like object related to the photo** and **will save it to the database** (and the object will be liked). In the end, we will **write a redirect function** that will redirect to the **last visited page** (**request.META['HTTP\_REFERER']**) and will **stop exactly at the photo we liked/unliked** (**f'#{photo\_id}'**):

Graphical user interface, text, application

Description automatically generated

Let us **refactor the template**. We will **implement the path** where the user should reach **when the heart button is clicked**. Then, the template will **check if the photo is connected to some of the Like objects**. Django uses "**like\_set**" to **reverse the search** - the Photo model is related to the Like model via One-to-Many relation; so we can get all like objects that are connected to the Photo model using the syntax "**like\_set.all**". In the same way, we can **count** **all** **likes for the photo**, this time using the method **count** in the template:

*...  
<!-- Start Like and Share Buttons -->*<div class="bottom">  
 <div class="actionBtns">  
 <div class="left">  
 *<!-- Start Like Button -->* <span class="heart">  
 **<a href="{% url 'like' photo.id %}">**

*<!-- if user has liked the photo -->*  
 **{% if photo.like\_set.all %}**  
 <svg style="color: red"  
 xmlns="http://www.w3.org/2000/svg"  
 width="24"  
 height="24"  
 fill="currentColor"  
 class="bi bi-heart-fill"  
 viewBox="0 0 16 16">  
 *<!-- Coordinate path -->* ...

*<!-- else -->*  
 **{% else %}**  
 <svg aria-label="Like"  
 color="#262626"  
 fill="#262626"  
 height="24"  
 role="img"  
 viewBox="0 0 48 48"  
 width="24">  
 **{% endif %}** *<!-- Coordinate path -->* ...  
 *<!-- End Like Button -->  
 ...*

*<!-- End Like and Share Buttons -->  
  
<!-- Number of Likes per Photo -->***<p class="likes">{{ photo.like\_set.count }} likes</p>**

*...*

One more thing we should do is to **add the photo id** to the template **in the photo div**. It is needed, so the **redirection** **works properly**:

*...  
<!-- Start Pet Photo -->*<div class="imgBx" id="{{ photo.id }}">  
 <img src="{% static 'images/axolotl.jpeg' %}" alt="post" class="cover">  
</div>  
*<!-- End Pet Photo -->*

*...*

### Implement Share Button Functionality

The **share button copies the photo details page URL in the clipboard**. To make the functionality, first, **add a path to a share view**: Graphical user interface, application, Word

Description automatically generated

There is an **additional module called pyperclip that we need to install**:Graphical user interface, text, application

Description automatically generated

**Import the** **copy()** **function from this module** in the **common/views.py file.** Then, we will **create a link to be copied** - the first half contains the **domain** (**request.META['HTTP\_HOST**'**]**) and the second half - the **path to the photo details page** (**resolve\_url('photo-details', photo\_pk)**). Finally, as in the **like\_functionality** view, we will **redirect the user to the last page visited on the exact photo they clicked**: Graphical user interface, text, application

Description automatically generated

Now, let us **refactor** **the pets-post.html** template. The only needed thing to do here is to **add the URL path**:

*...*

*<!-- Start Share Button -->***<a href="{% url 'share' photo.id %}">**  
 <svg aria-label="Share Post" class="\_8-yf5 " color="#262626" fill="#262626"  
 height="24" role="img" viewBox="0 0 48 48" width="24">  
 <path d="M47.8 3.8c-.3-.5-.8-.8-1.3-.8h-45C.9 3.1.3  
 3.5.1 4S0 5.2.4 5.7l15.9 15.6 5.5 22.6c.1.6.6  
 1 1.2 1.1h.2c.5 0 1-.3  
 1.3-.7l23.2-39c.4-.4.4-1 .1-1.5zM5.2  
 6.1h35.5L18 18.7 5.2 6.1zm18.7  
 33.6l-4.4-18.4L42.4 8.6 23.9 39.7z">  
 </path>  
 </svg>  
</a>  
 *<!-- End Share Button -->*

Let us **test the functionality**. Start the development server and **open the home page**. We should see a page like this:



Now, we **can click on the like button**, and it **should turn red**, and the **URL should change**. Now we have **1 like**: Graphical user interface, website

Description automatically generated

Next, let us click on the **share button**. Again, the page is reloaded, and if we **paste the URL**, it should look like this: "**127.0.0.1:8000/photos/1/"** and should **lead to the photo details page**.

### Add models to Pet Details Page

The **pet details page contains 2 main parts** - **pet personal data** and **pet photos**. It means that we should add the **Pet** object and all its photos to the **view's context**: Graphical user interface

Description automatically generated with medium confidence

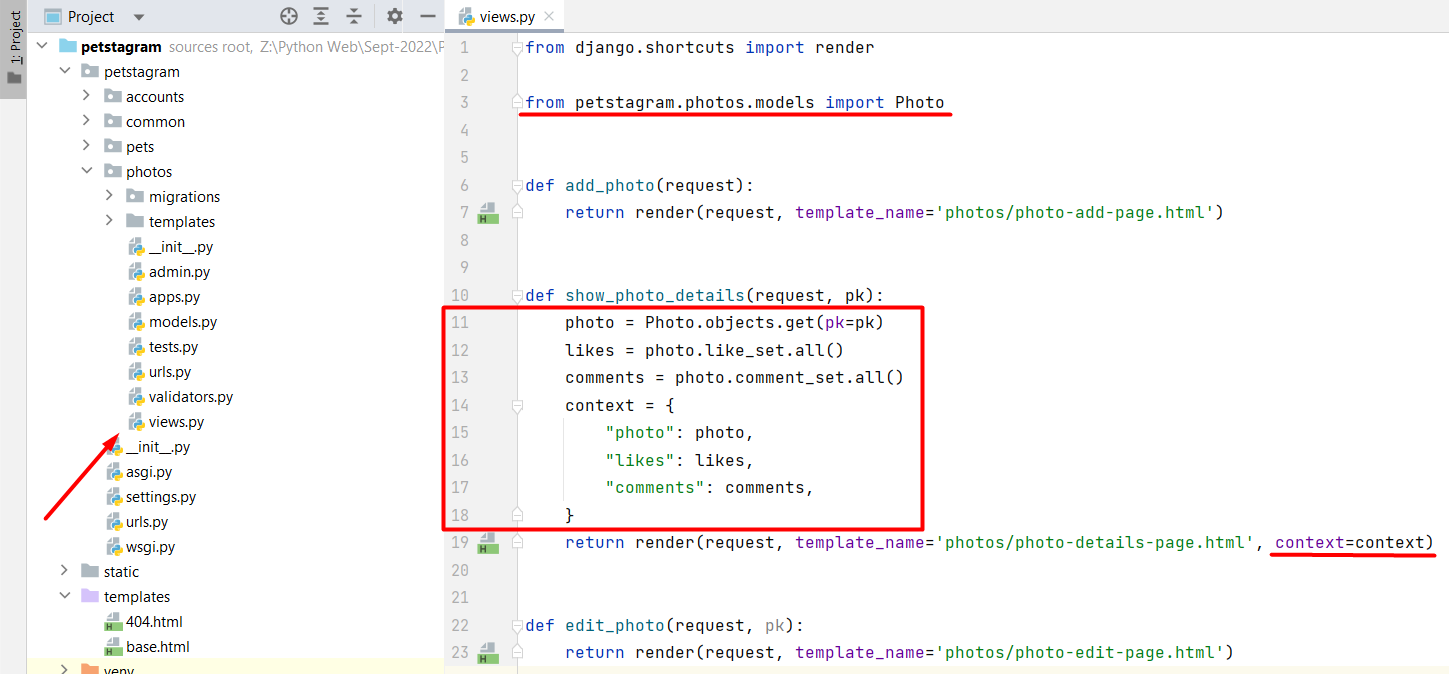
Next, let us **refactor the pet-details-page.html template**. We can **add the URL of the pet photo**, the **pet's name**, the **edit** and **delete** **buttons** paths, and **the total photos count**. We will add the **if statement** that checks if there are photos and **shows all photos of the pet**; otherwise, shows the **default no photos image**:

{% extends 'base.html' %}  
{% load static %}  
  
{% block content %}  
 <div class="pet-profile">  
 *<!-- Start Pet Personal Data Section -->* <div class="profile">  
 <div class="profile-data">  
 <div class="profile\_img">  
 <div class="image">  
 *<!-- Pet URL Image -->* **<img src="{{ pet.personal\_photo }}"**  
 alt="img8">  
 </div>  
 </div>  
 <div class="personal">  
 <div class="edit">  
 *<!-- Pet Name -->* **<p>{{ pet.name }}</p>**  
 *<!-- Pet Edit Button -->* **<a href="{% url 'edit-pet' "username" pet.slug %}">**  
 <img class="edit-img" src="/static/images/edit-pen-icon-6.jpg" alt="edit button">  
 </a>  
 *<!-- Pet Delete Button -->* **<a href="{% url 'delete-pet' "username" pet.slug %}">**  
 <img class="bin-img" src="/static/images/icon-remove-22.jpg" alt="bin button">  
 </a>  
 </div>

<div class="data">  
 *<!-- Pet Total Photos -->* **<span>{{ all\_photos.count }}</span>**  
 <p>photos</p>  
 </div>  
 </div>  
 </div>  
 </div>  
 *<!-- End Pet Personal Data Section -->* <div class="pet-posts">  
  
 **{% if all\_photos %}** {% include 'common/pets-posts.html' %}  
 *<!-- IF Photos End Pet Photos Post Section -->* **{% else %}** *<!-- IF NOT Photos Show No Post Image -->* <img class="no-posts" src="{% static '/images/no\_posts.png' %}" alt="no posts image">  
 **{% endif %}** </div>  
 </div>  
  
{% endblock %}

We **do not need to implement the photo posts context again** - it is already done. We just need to **use the same variable** **name** for all pet photos - **all\_photos**.

### Add models to Photo Details Page

Last for this workshop, we will **implement the models on the photo details page**. It consists of **Photo object** information, **photo likes,** and **comments** - so we need to **get the specific photo from the database**, **all its likes**, and **all its comments**, and **add it to the context**: 

Then, we **open the** **photo-details-page.html** **template** and we will **add the photo information**, **implement the like and share functionality**, and **add the number of likes for that photo**, **specify the tagged pets** and **date of** publication. And in the end, we will **add the comment object**, containing the **text**, and the **date and time of publication**:

{% extends 'base.html' %}  
*...* *<!-- Start Pet Photo Post Section -->  
 ...*  
 *<!-- IF the photo has location -->* **{% if photo.location %}  
 <span>{{ photo.location }}</span>  
 {% endif %}**  
 *<!-- IF the viewer is the creator of the photo -->* <div class="edit-delete-btns">  
  
 *<!-- Link to Edit Pet Photo Page -->***<a href="{% url 'edit-photo' photo.pk %}">**  
 <img class="edit-img" src="{% static

'/images/edit-pen-icon-6.jpg' %}"  
 alt="edit button">  
 </a>  
  
 *<!-- Link to Delete Pet Photo Page -->  
 ...*  
 *<!-- End User Details and Image Location Section -->  
 ...*  
 *<!-- Start Like and Share Buttons Section -->* <div class="actionBtns">  
 <div class="left">  
  
 *<!-- Start Like Button -->* <span class="heart">  
  
 *<!-- Link to Like Path -->* **<a href="{% url 'like' photo.id %}">**  
  
 *<!-- IF user has liked the photo -->* **{% if likes %}**  
 <svg style="color: red"  
 xmlns="http://www.w3.org/2000/svg"  
 width="24"  
 height="24"  
 fill="currentColor"  
 class="bi bi-heart-fill"  
 viewBox="0 0 16 16">  
 *<!-- Coordinate path -->  
 ...*  
 *<!-- IF NOT user has liked the photo -->* **{% else %}** <svg aria-label="Like"  
 color="#262626"  
 fill="#262626"  
 height="24"  
 role="img"  
 viewBox="0 0 48 48"  
 width="24">  
 **{% endif %}** *<!-- Coordinate path -->*

*...*

*<!-- Start Share Button -->  
  
 <!-- Link to Share Path -->***<a href="{% url 'share' photo.id %}">**  
 <svg...>  
 *...*  
 *<!-- End Like and Share Buttons Section -->  
  
 <!-- Number of Likes for the Photo -->* **<p class="likes">{{ likes.count }} likes</p>**  
  
 *<!-- Start Tagged Pets Section-->* **{% for pet in photo.tagged\_pets.all %}**  
 *<!-- Link to First Tagged Pet Details Page -->* **<a href="{% url 'pet-details' "username" pet.slug %}">**  
 <p class="message">  
 **<b>{{ pet.name }}</b>**  
 </p>  
 </a>  
 *<!-- End Tagged Pets Section-->* **{% endfor %}**  
 *<!-- Date of Publication or edit of the Photo -->* **<h5 class="postTime">{{ photo.date\_of\_publication }}</h5>**

*<!-- Start Comments Section -->*

**{% for comment in comments %}**

<div class="comments">  
 <div class="top">  
 <div class="userDetails">  
 <div class="comment-data">  
 <div class="profilepic">  
 <div class="profile\_img">  
 <div class="image">  
 *<!-- User Profile Image -->* <img src="{% static 'images/person.png' %}" alt="img8">  
 </div>  
 </div>  
 </div>  
 <p>  
 *<!-- Link to User Profile Details Page-->  
 <!-- User First and/or Last Name or username-->* <a href="">Steven Ivanov</a>  
 *<!-- User Comment -->* **{{ comment.text }}**

</p>  
 </div>  
 <span>**{{ comment.date\_time\_of\_publication** **}}**</span>  
 </div>  
 </div>  
 </div>  
 *<!-- End Comments Section -->*

**{% endfor %}** *...* *<!-- End Pet Photo Post Section -->* </div>  
 </div>  
  
{% endblock %}

### Order Comments

The comments do not appear to be in the order we want. The **last comment** published should **appear first** in the comment section. To do that we can use the model's **class Meta** option "**ordering**" to order the comments in **descending order** **by the date and time of publication**:Graphical user interface

Description automatically generated with medium confidence