Workflow:

The regular workflow in django is to create the model, then the view, then the url bound to the view, and lastly the template, the HTML file which the user will ultimately see. Then it all goes backwards, the user navigates on the HTML, and clicks an url, which is bound to a view and the view, it might also be bound to a model, and if it is, this model can affect the database by creating a new field, modifying it, or deleting it.

From the Database to the Template

TEMPLATE

URL

VIEW

MODEL

DATABASE

From the Template to the Database

This would be the normal workflow if we would decide to write to the database and read from it by using the simple functions <create> to create a new entry instead of using django forms. This works, but is not a programmatic way of doing it, and we would have to handle field by field manually. We have a better way to do it and that is through the django forms. This is another class which is imported from another module called forms and the workflow with this, is a little different.

The Requests module:

Obviously to send and receive the data, we ultimately rely on the **<requests>** **module**. From this module we know that we can connect, send data, and receive data from and to another machine media two main transference **methods**. These methods are the “GET” and the “POST” methods. There are other methods but for our forms we will only use those two because mainly, we will be:

* **showing** the form to the user or **“GET”** the HTML document, or,
* **getting** the data submitted to the form or **“POST”** from the HTML document.

Workflow with forms:

From the Database to the Template

FORM

MODEL

MODEL

VIEW

URL

TEMPLATE

DATABASE

From the Template to the Database

Forms:

Forms are essentially a way of getting data from the user via the HTML which is what we know as the front end of the app. A form is basically the same as a physic from, they have different fields to be filled, some of them are mandatory and some of them are not. The difference here is that with a digital form we have a lot of more options because we can deploy menus for the user to choose between a default set of options we want to restrict the user to choose from. Another feature of digital forms is that we can present them as simple or as complex as we want and also, if a field is mandatory, it won’t proceed to submit it.

<form action="/post/post" method= "POST" enctype="multipart/form-data">

        {% csrf\_token %}

        {{ form.as\_p }}

        <br><br>

        <input type="submit" value = "Submit" class="btn btn-primary">

</form>

(form example seen from the HTML file)

The Post Model

For the example covered in this document we’ll use the form from an app named post. Post is an app which mainly receives from the user three types of data, a [ Title, Text, Picture ]. This being said, there is a model also called Post (we write it like this because it is common to capitalize classes and a model is basically a class). Here is the model that we are currently using:

*§models.py*

from django.db import models

class Post(models.Model):

    title = models.CharField("Title", max\_length=110)

    text = models.TextField("Text")

    picture = models.ImageField("Picture", null = True, blank = True)

Bassically we are saying that our model will have three fields:

Title >>> a short text field with a controlled length.

Text >>> a long text field, or a text area, how is commonly known.

Picture >> here things get tricky because here we depend on the pillow library and **must** have it pre-installed in order to use this field. Also, we are saying that it doesn’t have to have a picture in the arguments

The strings that are taken as the first argument are the \_\_str\_\_ representation of each field.

Django uses a very clever idea to handle the getting data from the user. This idea is that **the data that we’ll receive in the FORM it is going to be the same as we have in our MODEL**.So, with this idea in mind, every form that we’ll need to fill our database it can be made out of a model.

This means that django comes with a bunch of features for the models which make very simple the way we retrieve the data form the user, and also the way that we shot the data to the user. For this well have to make a **forms.py** file in the same directory as the **views.py** from our app.

The PostModelForm:

The PostModelForm is another class, another model, but this time this model won’t be in the **models.py** file, neither will inherit from the models.Model class. It can work if we put it all in the same **models.py** file, but we won’t have an organized enviroment since we’ll have a **Post** model, and also a **PostForm** model. This is why we create the **forms.py** file. This file must be on the same folder as the **view.py** file because we will get this form and then we’ll pass it to the front just like that.

*§forms.py*

from .models import Post

from django import forms

class PostForm(forms.ModelForm):

    class Meta:

        model = Post

        fields = '\_\_all\_\_'

Here we are creating yet another model, but this time, this model will inherit from the **forms.ModelForm** class. Here the only thing we have to do is to define a class named **Meta** inside our **ModelForm** class. This class is the one that will contain the data that will be ultimately passed to the front end. The model variable will hold the model from which we want to make our form based in, and the fields variable holds how which fields do we want to actually form part of that form. In this case we chose the string “\_\_all\_\_” which of course means “send all the form fields”.

With this done we should have our form already made. The only thing we have left to do is to pass it to the front end’s context dictionary, which is that dictionary that we pass inside the arguments of the render function.

return render(request, 'post/post.html', {"entradas": entradas,},)

But instead, it is better to do it like this:

def post(request):

    entradas = Post.objects.all()[::-1]

    context = {"entradas": entradas}

    return render(request, 'post/post.html',context )

Widgets:

The widgets are another way to pass in formation to the front end. This information has to be passes inside the Meta class inside the ModelForm and it will be the object’s representation on the front end. We can use for that the same tags that we use on the front end, for example, we define a variable called attrs which holds a dictionary with the different attributes to be set inside that tag. We can then use any pre-defined attributes In the HTML or the CSS file to pass it down to the back-end. If the attribute is not defined for that field, this will raise an Error because you can’t use a bootstrap <’form-control’> class as an attribute for a field that is not defined for that specific tag, like a forms.ImageField.

*§forms.py*

class PostForm(forms.ModelForm):

    class Meta:

        model = Post

        fields = ('title', 'text', 'picture')

        widgets = {

            'title': forms.TextInput(attrs={'class':'form-control'}) ,

            'text': forms.Textarea(attrs={'class':'form-control'}),

            'picture': forms.FileInput(attrs={'class':'form-control'}),

            }

The class chosen is the default form class in bootstrap and this is how it looks:

Graphical user interface, application

Description automatically generated

'title': forms.TextInput(attrs={'class':'form-control'}) ,

'text': forms.Textarea(attrs={'class':'form-control'}),

'picture': forms.FileInput(attrs={'class':'form-control'}),

The from in the Front-End:

Now we know that our form is going to look really nice because we’ve bootsrapified it, but where is it? We have to pass it somehow to the front end and make it look like that. Well, now we must create a function that can:

* Show the HTML file with the empty form to be filled if we are just going there to see it (“GET”).
* Take the information sent by the user and save it to a variable.
* Validate the form. (Different types of fields have different validations.)
* Save all the data to the database
* Redirect us to the posts page to see our post

IMPORTANT

If you are new to requests, the GET and POST requests might confuse you and you might not quite understand how a function at the same time can, show you the “new-post” page on one mode (“GET”), and show you another page “posts” in another mode (“POST”). Also, why would “GET” show me a page where I “POST”, and “POST” show me the page where I see the information? That confusion is very common and that’s why I decided to explain it.

First of all, the function we are going to see next works by simple flow control using <if>, <else>, and <return>. The fact the <else> is not used might be confusing, but remember that inside a function, once you <return> something, the function blocks stop running. With this in mind, the function is written in a way where the most restrictive condition is written inside the <if> statement, and at the end, if all goes well, it <return>(s) a result, meaning that the function’s block stops running.

On the other hand, if the restrictive condition ( <if request.method == 'POST':> ) is not met, and it returns false, then the outside block of the function runs. This coding structure is convenient when we return something at the end of the block and we don’t want to explicitly write the else, but it might be confusing for who’s not used to it at first.

The next code has docstrings in it explaining each step of the process and what is the function of each line inside the function.

def new\_post(request):

    """Function that retrieves data from the form PostForm if the request method is POST and

also, this same function shows the page if the method is GET"""

    form = PostForm()

    if request.method == 'POST':

        """This part of the code runs if the method is POST. Then the form changes, this is

what we call a bound form, the same form can hold information, or it can be empty and

unbound."""

        form = PostForm(request.POST, request.FILES)

        """Once we pass to the form this arguments, the form is bound. Then it can do stuff

with this data. In this case the request.POST holds text and numbers, and

request.FILES holds the files, in this case is only one picture."""

        if form.is\_valid:

            """Different validations are run for different types of fields. Which validations

can be processes for each field is determined for the field type; the field types

are determined in the original model, not the form model."""

            form.save()

            """This line of code saves the data received from the user to the database. """

            return redirect("post-list")

        """The last line of code redirects us to the post.html file,

            where we can see our post."""

    """This part of the code is outside the code block we just was, and this code block is run

if the request method is other than POST. Why?

    The reason is that we have an <if> statement on the top, the most restrictive case

    is if the method is POST, if it is, the whole command block from before runs,

    if is not, the next command block runs, which are the last two lines in the whole code."""

    """Here we pass the form to the context dictionary which goes to the front end as

    a form object which the django tags language can understand and translate to HTML"""

    context = {'form':form }

    return render(request, 'post/new\_post.html', context)

The code we just saw is pretty simple and straight forward. It only looks long because of its docstrings but here is the actual code:

def new\_post(request):

    form = PostForm()

    if request.method == 'POST':

        form = PostForm(request.POST, request.FILES)

        if form.is\_valid:

            form.save()

            return redirect("post-list")

    context = {'form':form }

    return render(request, 'post/new\_post.html', context)

But this code strategy only works if we are passing data manually to the database, but what it we need to pass some data to the database, but this data, is not supposed to be filled by the user? Ok, What?

Let’s move on and see the next model so you can see what I’m talking about.