**Quick view of changes performed inside Django project**

Django-admin startproject PrivateSchool

*PrivateSchool/settings*

Inside the module settings we added the following code:

* INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'trainer.apps.TrainerConfig' , Added our app

'crispy\_forms' , Added crispy forms

]

* CRISPY\_TEMPLATE\_PACK = 'bootstrap4' Added between INSTALLED\_APPS AND MIDDLEWARE

In order to do the above we must install crispy forms with the following command :

pip install django-crispy-forms

*PrivateSchool/urls*

Inside the module urls we include the path of our app :

* urlpatterns = [

path('admin/', admin.site.urls), By default , we will not use it

path('' , include('trainer.urls')) , We need to import include

]

Django-admin startapp trainer

*trainer/*

Inside our app named trainer we added some new folders and modules.

First of all we created a folder named ‘templates’ and inside it a new folder with the same name as our app.This is where we will keep all our

Html files.

Likewise we made a new folder named ‘static’ and inside a new folder

named ‘css’.In ‘css’ we will keep all our css files.In ‘static’ folder we will keep js files , photos etc.

We created two new modules named ‘urls.py’ and ‘forms.py’.

Inside the modules urls which is the urls of our app we added :

* urlpatterns = [

path( '' , views.home, name='home')

path( 'add\_trainer/' , views.add\_trainer , name='add\_trainer'),

path('update\_trainer/<int:pk>/', views.update\_trainer, name='update\_trainer'),

path('delete\_trainer/<int:pk>/', views.delete\_trainer , name='delete\_trainer'),

path('show\_trainers/', views.show\_trainers, name='show\_trainers'),

]

We must import path and views.

When we type a specific url or get redirected , each path will look at the url and then it’s view.

*trainer/*

Inside the module forms we added :

* class TrainerForm(ModelForm):

class Meta:

model = Trainer

fields = '\_\_all\_\_'

Django provides a range of tools and libraries to build forms in order to accept input from users , process and then respond back.

This specific form will look directly to our model Trainer and takes all the fields as a parameter.

We need to import ModelForm.

Inside the module models we added :

* class Trainer(models.Model):

first\_name = models.CharField(max\_length = 50)

last\_name = models.CharField(max\_length = 50)

subject = models.CharField(max\_length = 50)

The model contains all the fields and behaviors of our stored data.

Each model is a Python class and each fields represents a database field.

For our case we gave three attributes and each one has a max length of fifty characters.

When we create new models in model.py the next thing we want to do is to inform django that we have a new model.

In order to do so we need to apply the next two commands :

* python manage.py makemigrations trainer
* python manage.py migrate trainer

Then we are good to go because our database will have a table.

Note that django provides us with an id automatically for each model we create .

*trainer/*

Inside module views we added all the necessary view functions that take a Web request and returns a Web response. It can be an html file , a redirect , an error , an XML document etc . In this project we need to implement CRUD functionality and the views help us to perform those operations.

To create an instance of model Trainer we added :

* def add\_trainer(request):

if request.method == "POST":

trainer\_form = TrainerForm(request.POST)

if trainer\_form.is\_valid():

trainer\_form.save()

return HttpResponseRedirect("/")

else:

trainer\_form = TrainerForm()

return render(request , "trainer/add\_trainer.html" , {"form" : trainer\_form})

View function add\_trainer checks if the method is POST or GET . If the method is GET we render our add\_trainer.html and we send also a dictionary which has the form we created in forms.py.

If the method is POST we take the data from the user and if it’s valid then we save. Because our form inherits ModelForm and is targeting our model Trainer the is\_valid() operation will check if the data are meeting our expectations according to the constrictions we made on our model.

(max\_length = 50)

*trainer/*

To update an instance of model Trainer we added :

* def update\_trainer(request , pk):

trainer = Trainer.objects.get(pk = pk)

form = TrainerForm(instance = trainer)

if request.method == 'POST':

form = TrainerForm(request.POST , instance = trainer)

if form.is\_valid():

form.save()

return HttpResponseRedirect("/show\_trainers/")

else:

formm = TrainerForm(instance = trainer)

return render(request ,

"trainer/update\_trainer.html" ,

{"form" : form} ,

)

In this view function we also have a parameter called pk which is the id of the model and must the same name with the url :

path('update\_trainer/<int:**pk**>/', views.update\_trainer, name='update\_trainer')

We create a dictionary to pass information later and we proceed to get the trainer with the given pk(primary key , id).

The instance is attribute will give us the specific model instance.

We continue like before according to whether the method is POST or GET.

If it is GET we fill the context dictionary with the trainer’s attributes so that we will be able to see it’s content before the update.

If it is POST we get the data and overwrite the trainer’s previous values with the new ones and then we redirect to the specific url.

*trainer/*

To delete an instance of model Trainer we added :

* def delete\_trainer(request , pk):

delete\_trainer = Trainer.objects.get(pk = pk)

delete\_trainer.delete()

messages.success(request , "Successfully deleted")

return HttpResponseRedirect("/show\_trainers/")

This view function also get the id of the specific trainer . In order to delete him we first need to ‘find’ him like before and then proceed with the delete() function .

To be able to perform update and delete we created a url which shows

all the trainer’s in our database and there you have the two option’s mentioned.

To see all the trainer’s in our database we added :

* def show\_trainers(request):

trainers\_list = Trainer.objects.all()

return render (request , "trainer/all\_trainers.html" ,

{"tr\_list" : trainers\_list}

)

The method all() returns a QuerySet of all the objects in the table trainers. Then we proceed to send it in a form of dictionary.

To use the model and the form we created we need to import them.

We also need to import render and HttpResponseRedirect.

As for the front end which is inside the folder templates make sure to have an editor that supports html files as well as the css files inside static folder.

My main goal was to introduce the changes inside the back end side of the project.