Django Application Design Document

# Starting up and setting the structure for the application

In order to set up a folder with the right Django file structure, the following command needs to be executed in the terminal.

django-admin startproject ‘name of application’

After this has been done, a Django folder structure is created. Lets assume that the name of the application is ‘ii\_app’, then I would enter in:

django-admin startproject ii\_website

The folder structure is now created, however, inside the structure, a developer can create multiple applications. In order to start creating an application, you need to navigate to the ii\_website folder, and then enter the following in the terminal:

python3 manage.py startapp “name of app”

So lets assume that the name of an application that I am building is named ii\_app, then the statement would look like this:

python3 manage.py startapp ii\_app

Once this is done, the folder structure for the app gets imbedded inside here.

# Views

A view allows the user to view a webpage. It does this by processing user requests. The job of a view inside the views.py file is to take a particular request, analyse it and then send back an appropriate response. A view is written inside the views.py file by writing a python function which takes in a request, and then returns a http response. Each view needs to be linked to a particular URL. A webpage can only be viewed if a URL exists for the website (view). In order to assign a URL to each view, you have to create a new file in the directory named url.py and then define the urls there, and link them to each view, so these URL patterns need to be defined.

Django has no obligation to execute the url patterns defined in the newly created urls.py file within the app structure. Rather, it goes to the urls.py file within the overall website structure and executes those urls. Therefore, you need to define the following code inside the urls.py file which resides inside the overall website structure.

urlpatterns = [

path('admin/', admin.site.urls),

path('ii\_app/', include('ii\_app.urls')),

]

The following library also needs to be imported:

from django.urls import include,path

You are telling the URL patterns to go to the path ii\_app, and then look through the ii\_app urls.

The request parameter is a HttpRequest object, which contains data about the request (see the [docs for django 3.2](https://docs.djangoproject.com/en/3.2/ref/request-response/)).

In your urls file, you are not calling the view.index function, just listing a reference to it. Django then calls the function when a matching request comes in and passes the HttpRequest object as a parameter.

# Settings.py

The databases dictionary contains information about the database engine (DBMS) being used to store the application data. In this instance, the database engine is SQLite, however, if you want to use a different database, you can install it and then name it in the databases dictionary. Secondly, the installed apps within the website are listed. Notice how the newly created app (i.e. ii\_app) is not listen in the settings.py file. Therefore, it needs to be added manually. It is added as the following format:

‘name\_of\_app.apps.name\_of`\_class\_in\_apps.py

# Models

Models allow us to create database tables. So models are the blueprints which can be used to create database tables.

These blueprints or models are created using Python classes and these models are created in models.py.

Every time an update has been made to the models, the following command updates the tables in the SQLite database:

python3 manage.py makemigrations ii\_app

This will create the models in the databases that have been defined in the models.py file. You will get something like this:

Migrations for 'ii\_app':

ii\_app/migrations/0001\_initial.py

- Create model Employee

0001 is the name of the model that has been created. However, this model needs to be migrated to the database now. This can be done with the following two command:

python3 manage.py sqlmigrate ii\_app 0001

python3 manage.py migrate

After doing this, the physical table has been created in the SQLite database.

Let’s say that we create a class named employees. We have to create an object from the employees’ class and then save the object using the save () method. The data then gets stored into the table.

If you want to add data into the database, you have to do it via python shell. First stop the server and then type in the following to open the shell:

python3 manage.py shell

We want to import the employees table into the python shell by writing:

from ii\_app.models import Employee

in order to see all of the data inside the table, you type the following into the shell:

Employee.objects.all()

This will give an empty queryset if there is no data in the table. In the shell, if we want to add data to the table, we have to create a new object, defining the variables within the class. This can be done as follows:

romi = Employee(employee\_name = 'Romi Dhillon', employee\_position = 'Data Architect', employee\_cost = 753)

A new object named romi has been created, but this object has only been created and not actioned to create anything in the database table. In order to store the object in the database, the following statement needs to be executed in the database:

romi.save()

You can get the unique id of this object by tying in the following into the shell

romi.id() or

romi.pk

# Django admin panel

Adding, removing or updating data from the python shell can be a tedious task. There is a quicker and more interative way of doing this, and it is by using the Django admin panel.

We have to create a super user first if we want to use the admin panel, and this is done with the following command:

python3 manage.py createsuperuser

We then have to set the username, email address, and password for this super user. Once the super user is created, you can login to the admin panel with these credentials. This is done through the browser. Open up the server, and in the url type in:

<http://127.0.0.1:8000/admin>

It will now ask for a username and password. Enter in the credentials. The admin.py file within the application structure must include the models/tables that you want to see in the admin panel. The models need to be registered to the admin site, so in the admin.py file, please type the following, with the name of the table passed into the register function:

admin.site.register(Employees)

The Employee model also needs to be imported into the admin.py file, so import the employee table from the models file with the following at the top of the admin.py file:

from .models import Employee

The model/table should now appear inside the admin panel so that it can be edited.

# Templates and Django Templating Language (DTL)

You first need to create a templates folder inside the application folder and then create a folder within the templates folder named the same name as your application. For example, you create a templates folder within ‘ii\_app’ and then create a folder named “ii\_app’ within the templates folder. All html templates such as home.html or about.html, etc, are placed inside the templates folder.

The views need to use the templates within the functions so that templates are rendered to the browser once the user wants to view the relevant page. Below is example code to explain how to do this

def resources(request):

employees = Employee.objects.all()

context = {

'employee\_list': employees,

}

return render (request, 'ii\_app/resources.html', context)

The return render function incorporates the request, so the function that is executed with the request inside it is read by the return render function. The return render function then takes in the appropriate template that needs to be rendered after the request is executed. The context is also taken in by the return render function as it needs to be used by the template to execute the templating language statements. Within the view (or resources function), there is a variable named employees which stores all entries in the Employee database. The context simply stores all of these items as a list, as the context is just a dictionary. The html template uses this dictionary item to loop through with the use of templating language. This is shown below.

{% for item in employee\_list %}

<ul>

<li>

{{item.name}} -- {{item.position}} -- {{item.day\_rate}} -- {{item.cf\_number}} -- {{item.project\_name}}

</li>

</ul>

{% endfor %}

By looping through each item in the employee list (specified as a key in the dictionary), it is now possible to render specific items from the database to the front-end.

If we want to create a page which displays information after clicking on a link, we can do that by creating another view which only shows information after a click of a link. Lets take a look at the view below.

def resource\_detail(request,employee\_id):

employee = Employee.objects.get(id = employee\_id)

context = {

'employee': employee,

}

return render (request, 'ii\_app/resource\_detail.html', context)

A http object request is taken in after the user has clicked on the url. The employee\_id is also taken in. The employee id is set to the object id of a table object (row). The context stores this information in a list, and names the key as employee. The request itself is rendered, and the template named resource\_detail is executed along with the context (which is used in the template).

The Django Templating Language is the default templating engine that Django uses. It has its own syntax. There are other engines such as Jinja.

# Static

The static folder holds information such as images, css files, JavaScript, etc. Inside the application folder (i.e. ii\_app), you have to create a static folder. Within that static folder, you have to create a folder with the same name as your application. For example, if the name of your application is ii\_app, then you have a folder within that named static. Within the static folder, you have a folder named ii\_app. The style.css file will be created within the ii\_app folder within the static folder.

Inside the html files, you need to add a link to the css with a link tag. The rel attribute within the link tag will be called stylesheet. The href attribute will be a direct link to the css file, but with the templating language. An example of this is below.

<link rel="stylesheet" href="% {static 'ii\_app.style.css'} %">

So you are looking up the static file within ii\_app and looking for style.css. At the top of the template html file, you need to type the following command in order to load the static files after they have been linked. This is done with:

{% load static %}

If the static files are still not working, a solution is available by going to the settings.py file and replacing STATIC\_URL = 'static/' with the below:

STATIC\_URL = 'static/'

STATICFILES\_DIRS= [

BASE\_DIR / 'static',

BASE\_DIR / 'ii\_app/static',

]

In order to allow the imageField function to work in models.py, the following code needs to be inserted into settings.py. This ensures that a root folder named ‘media’ is created after an image is inserted into the database. The url of where the image will be stored will start with /media in the browser.

MEDIA\_ROOT = 'media/'

MEDIA\_URL = '/media/'

Another piece of code also needs to be added to the website (not application) urls.py file. The code for this is found below:

urlpatterns = [

path('admin/', admin.site.urls),

path('ii\_app/', include('ii\_app.urls')),

]+static(settings.MEDIA\_URL,document\_root=settings.MEDIA\_ROOT)

This simply serves the media from the media root. The following libraries need to be inserted at the top of the urls.py file

from django.conf import settings

from django.conf.urls.static import static

If the CSS browser does not update after changes in the code, then press control, shift and R to refresh the browser.

# Base template / partials

# Previously, the navbar was included inside the templates. However, you can dedicate a template which is a partial just for specific components of the site. To do this, create a template named base.html and put the links for the CSS and Bootstrap in there. Put the

{% load static %} at the top of the base.html file and remove it from the other files, as the CSS is now linked to the base.html. Inside the other templates (other than the base.html), you have to put the following templating language code in them before and after the part you want to render. So before the body code, you put:

{%block body%}

After the body code you put:

{%endblock%}

At the top of these templates (all templates other than the base.html or partials), you have to put the following templating code:

{% extends 'ii\_app/base.html' %}

So lets assume there is a request for the resources page. The resources page is requested, and the first line is exectuted, which is that it is extended to base.html. The machine then looks through base.html and renders the base.html code. Then the block body code is searched for in the resources page, which is executed. In this manner, the navigation bar inside the base.html file is rendered first, and then the block body code is executed.

# Forms

In order to add forms to the application, the relevant urls and views need to be set up first. A new url pattern needs to be defined. An example is provided below.

path('addrisks/', views.risk\_form, name='risk\_form),

In the example above, a url pattern has been created with a path of addrisks and a view name of create risk.

The next step is to create a new file in the application directory named forms.py. This file will contain a class for each form that we are creating. You have to import the forms functions and the risks model into forms.py

from django import forms

from .models import Risk

The next step is to create a class which contains a meta class inside it. The meta class defines the fields that are going to be contained inside the form. Inside the meta class, we need to state the model that we are using and the fields that we want in the form (as a list). These are just the variables inside the respective model. Example code is provided below:

from django import forms

from .models import Risk

class RiskForm(forms.ModelForm):

class Meta:

model = Risk

fields = ["risk\_description","risk\_impact","risk\_probability",

"risk\_mitigation","risk\_owner","risk\_status","date\_opened"]

The next step is to create a template, so a new html template can be create with a suitable name such as risk\_form.html. Inside the HTML tag, we create a form tag and the method defined within the tag is the POST method, as we are posting the form data to the database. The csrf token needs to be included for security reasons. I can search more on this topic to find out more. A button has been included to submit the data for the form. The form data is being parsed into the html template to the {{form}} area from the views context. Example code is shown below:

<form action="POST">

{% csrf\_token %}

{{form}}

<button type="submit">Save</button>

</form>

We now must create a view to parse the form contents in the views.py file. An example and explanation of the view is provided below:

def risk\_form(request):

form = RiskForm(request.POST or None)

if form.is\_valid():

form.save()

return redirect('ii\_app/home.html')

return render (request, 'ii\_app/risk\_form.html', 'form':form)

The name of the view function should be the same as the one defined in the url pattern. A new object is initiated named form which calls the RiskForm class. The class takes in the post request, meaning that we are using the post request or we are using none. We next need to save the form data, but we first need to figure out if the data is valid. The is\_valid built in function checks if the form data is valid. After checking if it is valid, we want to save the form data in the database. We can then redirect the user to a new page, so they are directed to the chosen home page. We want to render the template next, by passing in the request itself, passing in the template name, and then passing in the context. For the context, we want to pass in the form object, which contains the posted data to the database which can be rendered to the html page.

The flow of the users form interaction is that Django will look for a view named risk\_form. So in the views.py file, the risk\_form view creates an object named form, which calls the class named RiskForm (which is defined in the forms.py file). Django then looks inside the RiskForm class and the meta class to see which model and fields/columns/attributes need to be used from the database. It will then create a new form for these fields within the chosen model. Django then looks inside the template named risk\_form to render information back to the user’s browser. The form is rendered to the browser. After the user submits the form, Django checks if the form data is valid, and then saves the data inside the database. The user is redirected. The following needs to be added to the top of the views.py file:

from django.shortcuts import render, redirect

from .forms import RiskForm

**Authentication – Sign Up functionality**

To authenticater users, you have to register them on your website by storing their username and password. In order to implement this authentication functionality, you have to create a completely new application, which can be called ‘users’.

To do this, you have to use the following command in the terminal:

Django-admin startapp users

The appropriate project structure and application should be in the folders and files.

Go into the apps.py file in the users app structure and copy the name of the app, which in this case is UsersConfig. The name of this application will need to be added to the INSTALLED\_APP list in the settings file of the application. It is added as:

‘users.apps.UsersConfig’

We are now free to go ahead and implement the authenticate functionality.

We have not now create a view to help the users to get registered. The views.py file in the users app needs to be used. We can create an build in form which comes with Django for sign in functionality. To import the built in form, we import the following at the top of the views.py file:

From django.contrib.auth.forms import UserCreationForm

We can now create a view and use this form in the view. The following code or view is created to render the register form back to the user. We pass in the form as the context.

def sign\_up (request):

form = UserCreationForm()

context = {

'form':form,

}

return render(request,'users/sign\_up.html',context)

In the directory, set up the following templates folder structure and then create a template name sign\_up.html.

Graphical user interface, text, application

Description automatically generated

Add the following basic code in the template which can be modified afterwards with bootstrap:

<body>

<form action="POST">

{%csrf\_token}

{{form}}

<button type="sign\_up"></button>

</form>

</body>

Next, go to the ii\_app directory and go to the urls.py file to add a path to register the user. Before doing this, the ii\_app and user app need to be connected. At the top of the file, add the following:

from users import views as user\_views

This is done as we need to import the views from the user’s application in order to render the view back to the user. We can now add a path to allow the user to sign\_up.

path('sign\_up/user\_views.sign\_up', name= 'sign\_up'),

The way this work is that:

* When the user clicks the sign-up button, the system goes to the user views to look for a sign\_up view inside the views.py file that is located inside the user app views.py file. Information is then rendered back to the user after executing the code located inside the sign-up.html file within the user application.

Whenever you submit the form data, it will be submitted by the POST method. Once the user has successfully signed up, they need to be redirected and a notification message needs to appear to show that they have signed up successfully. Information needs to be added to the sign-up view to make this happen. We can import pre-existing Django messages into the users views.py file. This is done by importing the following:

from django.contrib import messages

update the user views.py file with the following:

from django.shortcuts import redirect, render

from django.contrib.auth.forms import UserCreationForm

from django.contrib import messages

# Create your views here.

def sign\_up (request):

if request.method == 'POST':

form = UserCreationForm(request.POST)

if form.is\_valid():

form.save()

username = form.cleaned\_data.get('username')

messages.success(request,f'Hi {username}, you have successfully signed up to the Intelligent Infrastructure application')

return redirect('ii\_app/home')

else:

form = UserCreationForm()

context = {

'form':form,

}

return render(request,'users/sign\_up.html',context)

To explain the process, the first time the user accesses the sign-up page, the request method will not be POST as the user has not yet clicked the submit button, so has not posted any data to the database. The post method is shown in the template as well. Therefore, the system automatically goes past the if statement and goes to the else statement which renders the form to the user. Once the user fills out the form with data and presses submit, the if statement will be true, as a POST request will be made. The form is saved with the POST data inside it and then it is checked if it is valid (which is done with the is\_valid function for checks on password length, characters, etc). If the form is valid, a get request fetches the username from the form and the username is used inside the success notification message (messages.success). The user is then redirected to the ii\_app home page. The form.save( ) is used to store the details of the user.

**Authentication – Login functionality**

You first add the login path to the url patterns to the main ii site. The login functionality uses an in-built view in Django, so you don’t need to create a view for the login page. Import the following into the website:

from django.contrib.auth import views as authentication\_views

Put the following code in the url.py file.

urlpatterns = [

path('admin/', admin.site.urls),

path('ii\_app/', include('ii\_app.urls')),

path('sign\_up/',user\_views.sign\_up, name= 'sign\_up'),

path('login/',authentication\_views.LoginView.as\_view(template\_name = 'users/login.html'), name= 'login'),

path('logout/',authentication\_views.LogoutView.as\_view(template\_name = 'users/logout.html'), name= 'logout'),

]+static(settings.MEDIA\_URL,document\_root=settings.MEDIA\_ROOT)

The LoginView and logoutView is a class based view, which means that it is inherited from an existing class. In order to use these views, you have to use the as\_view( ) function. The login and logout views and paths have been created. These views still need templates to function. Inside the as\_view( ) function, you have to define the path where Django needs to look for the login template. Therefore, the template\_name has been defined above). After logging in, we can set where the user needs to be redirected to after logging in. In the settings.py file, add the following:

**Model inheritance**

We can create three different types of inheritance, which are as follows:

Abstract models

Multi-table model inheritance

Proxy models

To explain abstract model inheritance, see the code below:

A base class has been created which has the attributes named title, created, and updated. The meta class for this base class has been assigned the meta attribute of True. This class does not get created in the database, but it is only created so that all tables that are created in the database inherit the attributes that have been defined in the base class. ItemA, ItemB and ItemC all get created in the database. Notice how the class calls the baseItem and not models.Model. Class ItemA, ItemB and ItemC will all get created as tables in the database and will have the attributes defined in the base class, but will also have the attributes that have been defined within their respective classes.

Text

Description automatically generated

Multi-table model inheritance is different, as all tables are created in the database, but tables are built as an extension of others as a one-to-one connection. As the ISBN table is calling the Books table, the primary key of the books table is automatically linked to the ISBN id attribute.

Graphical user interface, text

Description automatically generated

**Aggregation**

Aggregation is used to retrieve values that are derived by summarising or aggregating a collection of objects. We select specific data that we want to retrieve from the database so that we can display or perform additional actions. Aggregation is about performing operations on objects after the data has been collected from the database. So such calculations can be average, sum, count, max, min, etc. So the way of looking at it is that we want to first perform a query to get the data, and then we want to use aggregate to perform the calculation.

**Annotation**

Annotate returns a new column or queryset which then holds the values of the query

**Django REST framework**

The Django REST framework is used to develop API’s. An API allows two systems to interact with one another. The REST acronym stands for Representational State Transfer which creates a representation of the data (copy of the data) on a website/server, and then sends the data across to another system/website. The REST framework allows CRUD operations on the representative data (but not the original data). An API is created by the developer to enable the communication of data between two systems.

When creating an API, the first action is to install the Django rest framework. This is done via the terminal:

pip3 install djangorestframework

The next step is to add the following in the settings.py file under installed apps

INSTALLED\_APPS = [

'rest\_framework',

'users.apps.UsersConfig',

'ii\_app.apps.IiAppConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

We need a way to get the data from tables and send it to other systems through the use of an API. Serialisation allows for complex data such as query sets or a model instance (a table) to be converted to native python data types. These data types can then easily be rendered into JSON or HTML or other content. So, in simple terms, the data in a table can be converted to JSON or HTML format by using serialisation. This is what an API actually does. So the next step is to go into the app folder (ii\_app) and create a new file in there named:

serialisers.py

Inside this file, serialisers are created. You need to import serialisers from the REST framework and the models/tables that you want to convert to JSON format.

from rest\_framework import serializers

from .models import Project

gary