Django

To install Django, run

```
pip3 install Django
```

To create a Django project, run

```
django-admin startproject PROJECT_NAME
```

To run the created web page

```
python manage.py runserver
```

Note that so far we have the default django web app,

Web app

to create an app,

```
python manage.py startapp hello
```

To add the new app, we have to register it to the **setting** file, under the `Installed Apps` section

```
# Application definition

INSTALLED_APPS = [
    'hello',
    'newyear',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Views

To create a view, we navigate to the **views.py**, and to create an instance of a view, we create a function

```
from django.http import HttpResponse
from django.shortcuts import render

# Create your views here.
def index(request):
    return HttpResponse("Hello, world!")
```

To display the view, we create its path in the **urls.py**, note we can create a **urls.py** for every app.

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.index, name="index"), #It contains 3 parts
]
#The first part is for the path url, note that in this case it is blank indicating
the default route
#The second part is for the location of the view
#The third part is for the name of the path, note that it is not necessary
```

To add the url to the global app, we navigate to the global urls.py and write the path to include.

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', include('hello.urls'))
]
```

We can also introduce place holders, in the route paths

```
#In the view.py

#Creating urls place holders
def greet(request, name):
    return HttpResponse(f"Hello, {name.capitalize()}!")

#in the urls.py
urlpatterns = [
    path('', views.index, name="index"),
    path("<str:name>", views.greet, name="greet"),
]
```

Templates

In the above example, we are only responding with **HttpResponse**, but to return a html page, we use templates

```
from . import templates

def index(request):
    return render (request, "hello/index.html")
```

The `hello/index.html` is the template that contains the html markup that we want.

Since the hello file is in the templates folder, we have to import the folder.

The templates in its self are html, hence we can not use variables in an html markup, but django has its own language to help us use variables. Similar to **blade** in laravel.

The `render` function can take three arguments,

- the request
- the template location
- the context, it is all the variables to be provided to the html

To demonstrate this, in the views.py

```
def greet(request, name):
    return render(request, "hello/greet.html", {
        "name": name.capitalize()
    } )
```

in the greet.html

```
<h1>Hello, {{name}}!</h1>
<p>{{name}} </p>
```

Conditional logic

```
<body>
    {% if newyear %}
        <h1>YES</h1>
    {%else%}
        <h1>NO</h1>
    {%endif%}
</body>
```

empty condition

It is used in conjunction with a for loop, where by incase the loop is empty, we can insert it

```
    <ul>
        {% for i in tasks %}
            <li>{{i}}</li>
        {% empty %}
            <li> No item </li>
        {% endfor %}
    </ul>
```

Static files

In django files that don't change can be referred to as static files, an example is CSS and they can be placed in the static directory.

```
{% load static %}

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Is it New Years</title>
    <link rel="stylesheet" href="{%static 'newyear/styles.css'%}">
</head>
<body>
    {% if newyear %}
        <h1>YES</h1>
    {%else%}
        <h1>NO</h1>
    {%endif%}
</body>
</html>
```

Tasks (Sample application)

Step 1 To create the app

```
python manage.py startapp tasks
```

Step 2 Register the app

In the settings.py in the parent app, add the **tasks** app to the lists of `Installed_apps`

```
# Application definition

INSTALLED_APPS = [
    'hello',
    'newyear',
    'tasks',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Step 3 Add the path

The path of the new app is to added into the urls.py of the parent app

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('hello/', include('hello.urls')),
    path('newyear/', include('newyear.urls')),
    path('tasks/', include('tasks.urls'))
]
```

Step 4 Create urls for the subsequent pages in the app

If the app is to contain additional pages, create a urls.py file inside the tasks app

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name="index"),
]
```

Step 5 Routing, function

The routing is done inside the views.py, by defining a function

```
from django.shortcuts import render
from . import templates

tasks = ["foo", "bar", "baz"]
# Create your views here.

def index(request):
    return render(request, "tasks/index.html", {
        "tasks": tasks
    })
```

Note, in this we have created a list `tasks` that contains sample items, and the list to be incorporated into the html as a variable, hence it is imported as shown.

Step 6 Creating the template

The html template is to created in templates/tasks subfolder.

```
{% load static %}

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Tasks</title>
</head>
<body>
    <ul>
        {% for i in tasks %}
            <li>{{i}}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Step 7 Test

At this step we can test to see if the html page will be displayed.

To do so we run the server

```
python manage.py runserver
```

Forms(Sample)

For the form example we are to continue on the tasks file, in that so far the page that we have created is for displaying the items in the list, to add elements into the list.

Step 8

Add the route function to the new page into views.py

```
def add(request):
    return render(request, "tasks/add.html")
```

Step 9 add url

For the new function we are to add its path to urls.py file

```
urlpatterns = [
    path('', views.index, name="index"),
    path('add', views.add, name="add"),
]
```

Step 10

Add a template of add.html

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Add Html</title>
</head>
<body>
    <h1>Add Tasks</h1>
    <form action="">
        <input type="text" name="task">
        <input type="submit">
    </form>
</body>
</html>
```

Step11 : Submitting form

Note, this part is after we have gone through `name space coalition`

We are to submit the form back to the **add url**, hence `<form action="{% url 'tasks:add' %}"` and the request method been **post** `<form action="{% url 'tasks:add' %}" method="post">`

```
<form action="{% url 'tasks:add' %}" method="post">
        <input type="text" name="task">
        <input type="submit">
</form>
```

`CSRF` Cross Site Request Forgery

Django has CSRF verification by default, to enable it in the form we add `{% csrf_token %}`

```
<form action="{% url 'tasks:add' %}" method="post">
     {% csrf_token %}
      <input type="text" name="task">
      <input type="submit">
</form>
```

To check on the token we can view the page source on the browser, it can be seen that an additional input field is added

`<input type="hidden" name="csrfmiddlewaretoken" value="CWlViA456blLmlKVdUorLzOW37ZLHLcKzQVPTvD54rBWWoYS4IWFZibjuWXdGV68">`

The input field is hidden, having values that change with each refresh.

Forms 2

Instead of hard-coding the html for the for we can instead, create a class in the views.py for the creation of the form and then in the html we import the class into the html file as a variable.

Step 1

Note that we will have to import the forms library in the views.py `from django import forms`

Step 2 Create the class

```
class NewTasksForm(forms.Form):
    task = forms.CharField(label="New Task")
```

Step 3 for the request function give the html access to the variable called task

```
def add(request):
    return render(request, "tasks/add.html", {
        "form": NewTasksForm()
    })
```

Step 4 To add another input field to the form,

```
class NewTasksForm(forms.Form):
    task = forms.CharField(label="New Task")
    priority = forms.IntegerField(label="Priority", min_value=1,max_value=10)
```

So far the views.py

```
from django.shortcuts import render
from . import templates
from django import forms
```

```
tasks = ["foo", "bar", "baz"]

class NewTasksForm(forms.Form):
    task = forms.CharField(label="New Task")
    priority = forms.IntegerField(label="Priority", min_value=1,max_value=10)

# Create your views here.

def index(request):
    return render(request, "tasks/index.html", {
        "tasks": tasks
    })

def add(request):
    return render(request, "tasks/add.html", {
        "form": NewTasksForm()
    })
```

Step 3 Incorporate the form into the html

```
<form action="{% url 'tasks:add' %}" method="post">
        {% csrf_token %}
        {{form}}
        <input type="submit">
</form>
```

Note that this method will introduce client side validation, in that the input will be validate by the client browser and no information is sent to the server.

Validation - Server -Side

To validate the data that is sent from the form to the server, we add the validation script to the view function that corresponds to the app. In our case, to validate the data from the form

```
def add(request):
    if request.method == "POST":  #We are checking if the request method is post and
if so cont...
        form = NewTasksForm(request.POST) #Save all the information submitted into
the form variable
        if form.is_valid():#If the form is valid, proceed
            task = form.cleaned_data['task'] #add the task item into a task variable
            tasks.append(task) #push the added task into the tasks lists
            return HttpResponseRedirect(reverse("tasks:index")) # to redirect the
user
        else:
        #If the form isnot valid we return the form back to them with all the fields
they had filled
            return render(request, "tasks/add.html", {
                "form": form
            })


    return render(request, "tasks/add.html", {
        "form": NewTasksForm()
    })
```

Template inheritance

It is a way to avoid repetitive html code, for instance in the task and add html files the body is the only part that is different, hence the rest can be placed in a common html file and the body to be substituted.

Step 1

Create the file that is to contain the part of the code that is similar as `layout.html`, under the templates folder

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Tasks</title>
</head>
<body>
    {%block body%}
    {%endblock%}
</body>
</html>
```

Note that, we use the `block` syntax to indicate where to replace the code, and in our case we have named the first block as `body`

Modifying the index.html

```
{% extends "tasks/layout.html" %}

{%block body%}

    <ul>
        {% for i in tasks %}
            <li>{{i}}</li>
        {% endfor %}
    </ul>

{%endblock%}
```

Modifying the add.html

```
{% extends "tasks/layout.html" %}

{%block body%}

    <h1>Add Tasks</h1>
    <form action="">
        <input type="text" name="task">
        <input type="submit">
    </form>

{%endblock%}
```

Note

`{% extends "tasks/layout.html" %}` is used to link to the layout template and we wrap the code we need to extend

inside the `{%block body%}....{%endblock%}`

Links

we can had code the url of the links

```
<a href="tasks/add"></a>
```

But, if we are to use the same link multiple times, we might loose track of the links in we are to change the links in future, hence we use of the name of the link as specified in the `urls.py` . Hence if we are to change the url, we change it only once in the urls.py file

```
<a href="{% url 'add' %}">Add New Task</a>
```

Name space coalition

This occurs when two things have the same name, for instance, in the urls.py we have the path named `index`, which is for both the tasks, hello and newyear apps, to eliminate the error we introduce `app_name` , we do this by giving each of the urls.py file a name

```
from django.urls import path
from . import views

app_name = "tasks"
urlpatterns = [
    path('', views.index, name="index"),
    path('add/', views.add, name="add"),
]
```

and as for the links

```
<a href="{% url 'tasks:add' %}">View Task</a>

<a href="{% url 'tasks:index' %}">View Task</a>
```

`tasks:index` states that from the app name `tasks` we use the root named `index`

Request Method

GET , it is a form of a request method, an example is when we type in a url, or press a link to go to another page, meaning we would like to get that particular page

POST , it is in turn used to send data back to the application, e.g while submitting forms

Sessions

So far so good, but there develops a problem in that, all the data is stored in a single global variable. In that, we differentiating between users is impossible. Sessions is in-turn used to eliminate this problem.

Sessions, work by recognizing who you are in each particular visit, and store data per each visit(session).

To take advantage of sessions in the tasks app, we will have to remove the global task variable, and an if clause is added to the index function, that is to create a `tasks` variable in a session.

```
def index(request):
    if "tasks" not in request.session:
        request.session["tasks"] = []

    return render(request, "tasks/index.html", {
        "tasks": request.session['tasks']
    })
```

While running the page it gives an errot `no such table: django_session` this is because **session** data is stored in tables (database).

Hence we are migrate the data to tables, on the terminal

```
python manage.py migrate
```

This creates default tables.

To add the **session task** variable, on the `add function`

```
def add(request):
    if request.method == "POST":
        form = NewTasksForm(request.POST)
        if form.is_valid():
            task = form.cleaned_data['task']
            #tasks.append(task) -- Initial without sessions
            request.session["tasks"] += [task] #With sessions
            return HttpResponseRedirect(reverse("tasks:index"))
        else:
            return render(request, "tasks/add.html", {
                "form": form
            })


    return render(request, "tasks/add.html", {
        "form": NewTasksForm()
    })
```