What Is a Framework????

**Framework** –S/w framework is a platform for developing software applications. It already have predefined classes functions that can be used to process input , manage hardware devices and interact with System softwares.

**Django** is an extremely popular and fully featured server-side web framework, written in Python. It provides a set of tools and functionalities that solves many problems associated with web development such as user sign up, Authorization ,Database Access security features, template design ,URL Mapping and much more. Django helps us to develop secure and reliable web application in quick and standardized way without having to reinvent the wheel.

**Installation**

The first thing we need to do is install some programs on our machine so to be able to start playing with Django. The basic setup consists of installing **Python**, **Virtualenv**, and **Django**

Using virtual environments is not mandatory, but it’s highly recommended. When developing Web sites or Web projects with Django, it’s very common to have to install external libraries to support the development. Using virtual environments, each project you develop will have its isolated environment. So the dependencies won’t clash. It also allows you to maintain in your local machine projects that run on different Django versions.

### **What is the Django development environment?**

The development environment is an installation of Django on your local computer that you can use for developing and testing Django apps prior to deploying them to a production environment.

The main tools that Django itself provides are a set of Python scripts for creating and working with Django projects, along with a simple development webserver that you can use to test local (i.e. on your computer, not on an external web server) Django web applications on your computer's web browser.

**Creating A Project**

First open a command prompt/terminal, navigate to where you want to store your Django apps (make it somewhere easy to find like inside your *documents* folder), and create a folder for your new website (in this case:*django\_projects*). Then enter into the folder using the cd command:

mkdir django\_projects

cd django\_projects

Create the new project using the django-admin startproject command as shown, and then navigate into the folder.

django-admin startproject mypro

cd mypro

The django-admin tool creates a folder/file structure as shown below:

*mypro/*

  manage.py

*mypro/*

\_\_init\_\_.py

    settings.py

    urls.py

    wsgi.py

Our current working directory should look something like this:

../django\_projects/mypro/

The *mypro* project sub-folder is the entry point for the website:

* **\_\_init\_\_.py**is an empty file that instructs Python to treat this directory as a Python package.
* **settings.py** contains all the website settings. This is where we register any applications we create, the location of our static files, database configuration details, etc.
* **urls.py** defines the site url-to-view mappings. While this could contain *all*the url mapping code, it is more common to delegate some of the mapping to particular applications.
* **wsgi.py** is used to help your Django application communicate with the web server.  This file is a simple gateway interface used for deployment.

The **manage.py** script is used to create applications, work with databases, and start the development web server.

We can run the development web server from within this folder using **manage.py** and the runserver command, as shown.

$ python manage.py runserver

Performing system checks...

System check identified no issues (0 silenced).

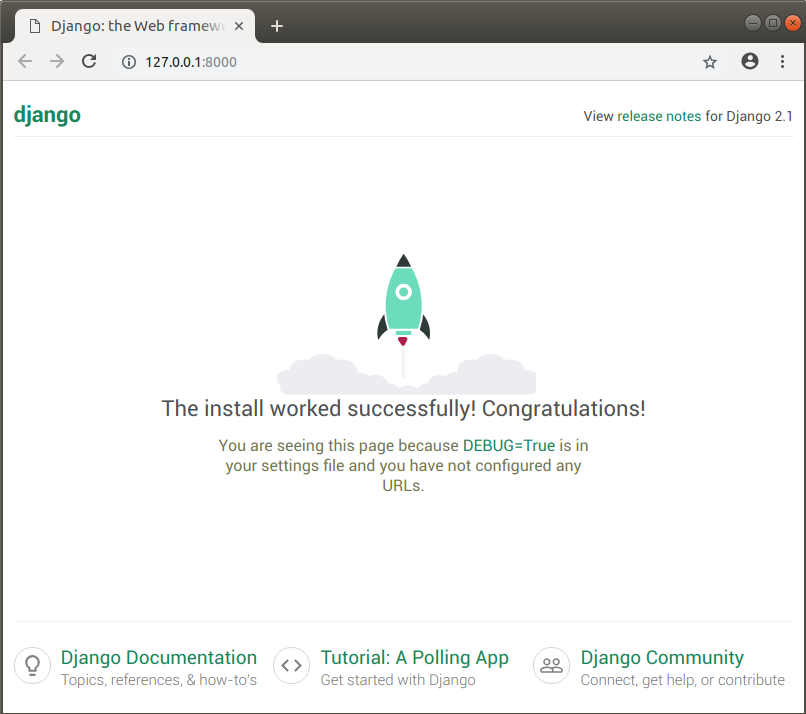
You have 15 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.

Run 'python manage.py migrate' to apply them.

Django version 2.1.2, using settings 'mytestsite.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CONTROL-C.

Once the server is running you can view the site by navigating to the following URL on your local web browser: http://127.0.0.1:8000/. You should see a site that looks like this:  


**Django Apps**

In the Django philosophy we have two important concepts:

* **app**: is a Web application that does something. An app usually is composed of a set of models (database tables), views, templates, tests.
* **project**: is a collection of configurations and apps. One project can be composed of multiple apps, or a single app.

It’s important to note that you can’t run a Django **app** without a **project**

## **Creating A Application**

Next, run the following command to create the myapp application that will live inside our mypro project (this must be run in the same folder as your project's **manage.py**):

python manage.py startapp myapp

The tool creates a new folder and populates it with files for the different parts of the application (shown in bold below). Most of the files are usefully named after their purpose (e.g. views should be stored in **views.py**, models in **models.py**, tests in **tests.py**, administration site configuration in **admin.py**, application registration in **apps.py**) and contain some minimal boilerplate code for working with the associated objects.

The updated project directory should now look like this:

*mypro/*

  manage.py

*mypro/*

***myapp/***

**admin.py**

**apps.py**

**models.py**

**tests.py**

**views.py**

**\_\_init\_\_.py**

***migrations/***

**migrations**:-Keeps track of changes if you create in the models so to keep db and models synchronized.

**\_\_init\_\_.py**:-  an empty file created here so that Django/Python will recognise the folder as a [Python Package](https://docs.python.org/3/tutorial/modules.html#packages)

**admin.py**:-Django-admin configuration files

**apps.py:-**apps specific configuration file

**models.py**:-Defines entities in web application. The models are translated automatically by Django into database tables.

**tests.py**:-Unit test for the app

**views.py**:-Request/response cycle of our web application. Each view of our Application is defined inside view file.

Now that we created our first app, let’s configure our project to *use* it.

To do that, open the **settings.py** and try to find the INSTALLED\_APPS variable:

**settings.py**

INSTALLED\_APPS **=** [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

As you can see, Django already come with 6 built-in apps installed. They offer common functionalities that most Web applications need, like authentication, sessions, static files management (images, javascripts, css, etc.) and so on.

But for now just add our **myapp** app to the list of INSTALLED\_APPS:

INSTALLED\_APPS **=** [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'myapp',

]

Let’s write our first **view**.

#### Hello, World!

.

But for now, let’s just experiment how it looks like to create a new page with Django.

Open the **views.py** file inside the myapp app, and add the following code:

**views.py**

**from django.http import HttpResponse**

**def home(request):**

**return HttpResponse('Hello, World!')**

Views are Python functions that receive an HttpRequest object and returns an HttpResponseobject. Receive a request as a parameter and returns a response as a result.

So, here we defined a simple view called **home** which simply returns a message saying **Hello, World!**.

Now we have to tell Django *when* to serve this view. It’s done inside the **urls.py** file:

**urls.py**

**from django.conf.urls import url**

**from django.contrib import admin**

**from myapp import views**

**urlpatterns = [**

**url(r'^$', views.home, name='home'),**

**url(r'^admin/', admin.site.urls),**

**]**

If you compare the snippet above with your **urls.py** file, you will notice I added the following new line:url(r'^$', views.home, name='home') and imported the **views** module from our app myapp using from myapp import views.

But for now, Django works with **regex** to match the requested URL. For our **home** view, I’m using the ^$ regex, which will match an empty path, which is the homepage (this url: **http://127.0.0.1:8000**). If I wanted to match the URL **http://127.0.0.1:8000/homepage/**, my url would be: url(r'^homepage/$', views.home, name='home').

Let’s see what happen:

python manage.py runserver

In a Web browser, open the http://127.0.0.1:8000 URL:

We can see Hello World in the browser….

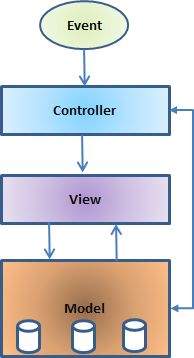
**MVC MODEL**

 Django is a Python web framework. And like most modern framework, Django supports the MVC pattern. Applications that provides UI (web or desktop), usually based on MVC architecture. MVC pattern is based on three components: Model, View, and Controller.

**M**odel **V**iew **C**ontroller or **MVC** as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts −

* **Model** − The lowest level of the pattern which is responsible for maintaining data.
* **View** − This is responsible for displaying all or a portion of the data to the user.
* **Controller** − Software Code that controls the interactions between the Model and View.

MVC is popular as it isolates the application logic from the user interface layer and supports separation of concerns. Here the Controller receives all requests for the application and then works with the Model to prepare any data needed by the View. The View then uses the data prepared by the Controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.



**DJANGO MVC - MVT Pattern**

The Model-View-Template (MVT) is slightly different from MVC. In fact the main difference between the two patterns is that Django itself takes care of the Controller part (Software Code that controls the interactions between the Model and View), leaving us with the template. The template is a HTML file mixed with Django Template Language (DTL).

The following diagram illustrates how each of the components of the MVT pattern interacts with each other to serve a user request −



The developer provides the Model, the view and the template then just maps it to a URL and Django does the magic to serve it to the user.

The Django flow goes something like this:

1) Visitor’s browser asks for a URL.

2) Django matches the request against its urls.py files.

3) If a match is found, Django moves on to the view that’s associated with the URL. Views are generally found inside each app in the views.py file.

4) The view generally handles all the database manipulations(models). It grabs data and passes it on.

5) A template (specified in the view) then displays that data.

Previously we use a direct mapping from urls.py file to views.py file. Now we are adding an additional function **include()** importing from **django.conf.urls.**

**Include()** function allows us to look for a match with regular expression and link back to our applications url file. But we have to manually add urls.py file in each application files.

So we would add project urls.py file like this.

**urls.py**

from django.conf.urls import url,include

from django.contrib import admin

from app1 import views #for home view

urlpatterns**=** [

url(r'^$', views**.**home, name**=**'home'),

url(r'^admin/', admin**.**site**.**urls),

url(r’^app1', include(‘app1.urls’)),

url(r’^app2', include(‘app2.urls’)),

]

**url(r’^app1', include(‘app1.urls’))** this would allows to look for any url that has the pattern 127.0.0.1:8000/app1 if django gets that match,the include() function tells Django to go look at the urls.py inside of the app1 folder.

Instead of listing all the urls in the main url file, we set the reference to each application.

**Templates**

Being a web framework, Django needs a convenient way to generate HTML dynamically. The most common approach relies on templates. A template contains the static parts of the desired HTML output(parts that are always the same) as well as some special syntax describing how dynamic content will be inserted.

In HTML, we can't really write Python code, because browsers don't understand it. They know only HTML. We know that HTML is rather static, while Python is much more dynamic.

**Django template language a**llow us to transfer Python-like things into HTML, so you can build dynamic websites faster. The [Django template language](https://docs.djangoproject.com/en/2.2/ref/templates/language/) is Django’s own template system. Django template uses its own syntax to deal with variable, tags, expressions etc. These template variables will allow us to inject content into the HTML directly from django.

**Variables**

Variables associated with a context can be accessed by {{}} (double curly braces). For example, a variable name value is rahul. Then the following statement will replace name with its value.

1. My name is {{name}}.(in html)
3. My name is rahul  (in webpage)

**Tags**

In a template, Tags provide arbitrary logic in the rendering process. For example, a tag can output content, serve as a control structure e.g. an "if" statement or a "for" loop, grab content from a database etc.

Tags are surrounded by {% %} braces. For example.

1. {% csrf\_token %}
3. {% **if** user.is\_authenticated %}
4. Hello, {{ user.username }}.
5. {% endif %}

**Configuration**

To get started with templates we first need to create templates directory inside our container folder. Next step is to let Django know of the templates by editing DIR key inside of the TEMPLATES directory in the settings.py file.

We want our Django project to be easily transferable from one computer to another, but the DIR key will require a “hard-coded” path.

So for this, First we have to find the path to the templates folder. We can use Python’s os module to dynamically generate correct file path,strings, regardless of the computer.

Under the project folder select settings.py file.

We can see that import os statement in the beginning

Next is

BASE\_DIR= os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))) .it builds the project path

Now we can use this BASE\_DIR to add templates.

TEMPLATE\_DIR=os.path.join(BASE\_DIR,”templates”)# This is the path to the template folder

Templates engines are configured with the [**TEMPLATES**](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES) setting. It’s a list of configurations, one for each engine. The default value is empty.

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [],

'APP\_DIRS': **True**,

'OPTIONS': {

*# ... some options here ...*

},

},

]

[**DIRS**](https://docs.djangoproject.com/en/2.2/ref/settings/#std:setting-TEMPLATES-DIRS) defines a list of directories where the engine should look for template source files, in search order.

Add TEMPLATE\_DIR to the DIRS.

**RENDERING TEMPLATE**

Django makes it possible to separate python and HTML, the python goes in views and HTML goes in templates. To link the two, Django relies on the render function and the Django Template language.

**The Render Function**

This function takes three parameters −

* **Request** − The initial request.
* **The path to the template** − This is the path relative to the TEMPLATE\_DIRS option in the project settings.py variables.
* **Dictionary of parameters** − A dictionary that contains all variables needed in the template.

**Django Template Simple Example**

First, create a directory **templates** inside the project. Create a index.html file inside templates folder

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <title>Index</title>
6. </head>
7. <body>
8. <h2>Welcome to Django!!!</h2>
9. </body>

</html>

Create a function in Views.py file inside application for rendering template index.html. for eg.

def index(request):

render(request,’index.html’)

link urls to index function inside views file like this.

url(r’^index,views.index,name=”index”)

Passing values to the html file from views.py file

For eg:

Inside Views.py file

def index(request):

d={“insert\_me”:”I am in Django”}

return render (request,’index.html,context=d)

Inside index.html add template tags to get the value.

{{insert\_me}}

When the page corresponding to the index view loads, then This will displays the value I am in Django.

**STATIC FILES**

In django the additional files such as images, JavaScript, or CSS refer to as “static files.

**Configuration**

To get started with static **create a new folder “static” in the root of the main project***same level as the “templates” directory****.***

Next step is to let Django know of the static by adding path. You can use STATICFILES\_DIRS for that:

Specify STATICFILES\_DIRS In Django Project settings.py File.

First find the path to the static folder and then add it to the STATICFILES\_DIRS.

STATIC\_DIR=os.path.join(BASE\_DIR,”static”)

STATICFILES\_DIRS = [ STATIC\_DIR,]

Create a **css** ,image and js sub folder in the **static** folder, then you should save your image file ,css file or js file in them accordingly.

Adding html files to the template

Below is the steps:

1. Use {% load staticfiles %} at first to load django static tag.
2. Use django static tag to reference the static files.

**<script** src**=**"{% static 'js/click.js' %}"**></script>**

**<link** rel**=**”stylesheet” href**=**”{% static 'css/style.css' %}”**>**

**<img** src**=**"{% static 'image/click.js' %}"**></script>**

**MODEL**

A model is a class that represents table or collection in our DB, and where every attribute of the class is a field of the table or collection. Models are defined in the app1/models.py. We will be using a SQLite database to store our data. This is the default Django database .

Following is a model created as an example −

from django.db import models

class preson(models.Model):

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

age = models.IntegerField()

def \_\_str\_\_(self):

return self. name

Every model inherits from django.db.models.Model.

Our class has 2 attributes (1 CharField and 1 IntegerField), those will be the table fields.

**Registering models**

Django comes with a built-in admin interface. When you ran startproject , Django created and configured the default admin site for you. – with Django’s admin you can authenticate users, display and handle forms and validate input; all automatically. Django also provides a convenient interface to manage model data.

The Django admin *application* can use your models to automatically build a site area that you can use to create, view, update, and delete records. This can save you a lot of time during development, making it very easy to test your models and get a feel for whether you have the *right*data.

First, open **admin.py** in the application.It currently looks like this — note that it already imports django.contrib.admin:

from django.contrib import admin

# Register your models here.

Register the models by copying the following text into the bottom of the file. This code simply imports the models and then calls admin.site.register to register each of them.

from app1.models import person

admin.site.register(person)

**Create tables for models in your database**

The last step here is to add our new model to our database. First we have to make Django know that we have some changes in our model. (We have just created it!) Go to our console window and type **python manage.py makemigrations**.

* [**makemigrations**](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-makemigrations), which is responsible for creating new migrations based on the changes you have made to your models.Migrations are Django’s way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema
* Django prepared a migration file for us that we now have to apply to our database.

There is a column for each field and an additional column id for the primary key, which Django creates automatically unless you explicitly specify a primary key in your model.

## **How to apply a migrations**

Once the migration, we need to apply them, so that changes can be reflected in the databases as well.

To apply migration, we need to run migrate command as follows

**python manage.py migrate**

* [**migrate**](https://docs.djangoproject.com/en/2.2/ref/django-admin/#django-admin-migrate), which is responsible for applying and unapplying migrations.

By running this command, all the unapplied migration are applied to the database.

**Creating a superuser**

In order to log into the admin site, we need a user account with *Staff* status enabled. In order to view and create records we also need this user to have permissions to manage all our objects.  You can create a "superuser" account that has full access to the site and all needed permissions using **manage.py**.

Call the following command, in the same directory as **manage.py**, to create the superuser. You will be prompted to enter a username, email address, and *strong* password.

**python manage.py createsuperuser**

Enter your desired username and press enter.

Username: user1

You will then be prompted for your email address:

Email address: a@example.com

The final step is to enter your password. You will be asked to enter your password twice, the second time as a confirmation of the first.

Password: \*\*\*\*\*\*\*\*\*\*

Password (again): \*\*\*\*\*\*\*\*\*

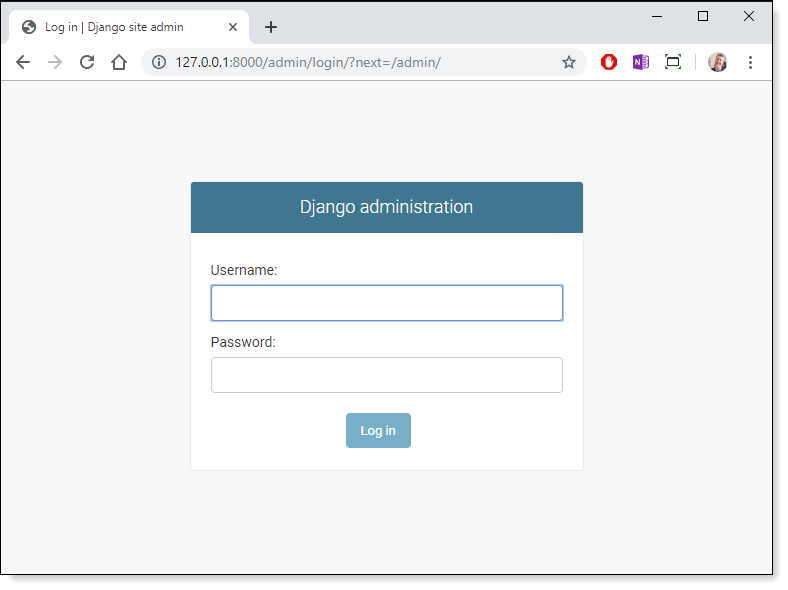
Superuser created successfully.

Once this command completes a new superuser will have been added to the database. Now restart the development server so we can test the login:

**python manage.py runserver**

**Logging in and using the site**[**Section**](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Admin_site#Logging_in_and_using_the_site)

To login to the site, open the */admin* URL (e.g. [http://127.0.0.1:8000/admin](http://127.0.0.1:8000/admin/)) and enter our new superuser userid and password credentials (we'll be redirected to the *login* page, and then back to the */admin* URL after you've entered your details).



This part of the site displays all our models, grouped by installed application. You can click on a model name to go to a screen that lists all its associated records, and you can further click on those records to edit them. You can also directly click the **Add** link next to each model to start creating a record of that type.

At the top of the index page is the **Authentication and Authorization** group with two types of editable content: **Groups** and **Users**. They are provided by the authentication framework included in Django

Click on the **Add**link to the right of persons to create a new person. Enter values for the name and age fields. When you're done you can press **SAVE**, **Save and add another**, or **Save and continue editing** to save the record.

When you've finished adding persons, click on the **Home** link in the top bookmark to be taken back to the main admin page. Then click on the **persons** link to display the current list of persons . Now that you've added a few persons.The name of the each person is displayed; this is the value returned in the person model's \_\_str\_\_() method that we specified in the model definition.

From this list you can delete persons by selecting the checkbox next to the persons you don't want, selecting the *delete...* action from the *Action* drop-down list, and then pressing the **Go** button.

Now we added some persons details to the table.

In order to display the persons, we need to create a view to list all the persons or filter the persons by selected category. Edit **views.py** file of our application and make sure it has this code:.

**Views.py**

from django.shortcuts import render

from django.http import HttpResponse

from app1.models import person

def per(request):

user\_list = person.objects.all()

return render(request,'per.html',{"users":user\_list})

**per.html**

<!DOCUMENT html>

<html>

<head>

<title>Employee</title>

</head>

<body>

<h1>Here are the persons</h1>

{% if users %}

<ol>

{% for e in users %}

<li>Person Info</li>

<ul>

<li>Name: {{e.name}}</li>

<li>Age: {{e.age}}</li>

</ul>

{% endfor %}

</ol>

{% endif %}

</body>

</html>

**Link to the views file**

**urls.py**

from django.conf.urls import url,include

from django.contrib import admin

from app1 import views #for home view

urlpatterns**=** [

url(r'^$', views**.**home, name**=**'home'),

url(r'^admin/', admin**.**site**.**urls),

url(r’^per', views**.**per, name**=**'per'),

]

**FORMS**

An [**HTML Form**](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms) is a group of one or more fields/widgets on a web page, which can be used to collect information from users for submission to a server. Forms are a flexible mechanism for collecting user input because there are suitable widgets for entering many different types of data, including text boxes, checkboxes, radio buttons, date pickers, etc. Forms are also a relatively secure way of sharing data with the server, as they allow us to send data in POST requests with cross-site request forgery protection.

While we haven't created any forms so far, we've already encountered them in the Django Admin site.

  Developers need to write HTML for the form, validate and properly sanitise entered data on the server (and possibly also in the browser), repost the form with error messages to inform users of any invalid fields, handle the data when it has successfully been submitted, and finally respond to the user in some way to indicate success. **Django Forms** take a lot of the work out of all these steps, by providing a framework that lets you define forms and their fields programmatically, and then use these objects to both generate the form HTML code and handle much of the validation and user interaction.

#### Declaring a Form

The nice thing about Django forms is that we can either define one from scratch or create a ModelForm which will save the result of the form to the model.

The declaration syntax for a Form is very similar to that for declaring a Model, and shares the same field types (and some similar parameters). This makes sense because in both cases we need to ensure that each field handles the right types of data, is constrained to valid data, and has a description for display/documentation.

Form data is stored in an application's forms.py file, inside the application directory. To create a Form, we import the forms library, derive from the Form class, and declare the form's fields. **A very basic form class is shown below —**

from django **import** forms

from app1.models import person

**class** PersonForm(forms.Form):

    name = forms.CharField(max\_length=150)

    age  = forms.integerField()

ModelForm is a class which is used to create an HTML form by using the Model. It is an efficient way to create a form without writing HTML code .Django automatically does it for us to reduce the application development time. For example, suppose we have a model containing various fields, we don't need to repeat the fields in the form file.

For this reason, Django provides a helper class Meta which allows us to create a Form class from a Django model.

from django **import** forms

from app1.models **import** person

**class** PersonForm(forms.ModelForm):

**class** Meta:

        model =person

        fields = "\_\_all\_\_"

Above is the PersonForm class that represents the form and which inherits its behavior from the django.forms.ModelForm class, the last of which is what makes it a model form. Notice the PersonForm class lacks any form fields like those listed above form class, instead it declares a Meta class section like the one used in models.

The Meta class section for the PersonForm specifies two options: model and fields. The model option indicates which model to use to generate the form, in this case, the person model. The fields option indicates which model fields to use to generate the form, in the case, '\_\_all\_\_' tells Django to use all the model fields in the model.

Django doesn't expect the structure of a model to fit perfectly with a form, to the point Django also expects you to explicitly tell it which fields of the backing model should or shouldn't become part of the model form. This is achieved with either the fields option -- to indicate which model fields become part of the model form -- of the excludes option -- to indicate which model fields shouldn't become part of the model form. The fields or exclude option is always required, even when a model form will contain all fields of the backing model.The option  fields='\_\_all\_\_' to create a model form that captures the same set of fields as its backing model.

fields =(name) indicates to create a model form with only name as field

excludes=[name] indicates to create a model form with only age as field and excludes name filed.

#### Django model form processing

**views.py method to process model form**

def persons(request):

if request.method == 'POST':

# POST, generate bound form with data from the request

form = PersonForm(request.POST)

# check if it's valid:

if form.is\_valid():

# Insert into DB

**form.save()**

# redirect to a new URL:

return render(index(request))

else:

# GET, generate unbound (blank) form

form = PersonForm()

return render(request,'persons.html',{'form':form})

The most important construct in the view method is the if/else condition that checks the request method type. If the request method type is POST (i.e. data submission) the form's data is processed, but if the request method type is anything else (i.e. initial request ) an empty form is generated to produce the same behaviour. Notice the last line is a return statement that assigns the form instance -- whether empty(a.k.a. unbound) or populated (a.k.a. bound) -- and returns it to a template for rendering..

Now let's take a closer look at the POST logic.If the request method type is POST it means there's incoming user data, so we access the incoming data with the request.POST reference and initialize the Django form with it. using request.POST as the argument of a Django form class is sufficient to populate a Django form instance with user data.It's worth mentioning that a Django form instance created in this manner (i.e. with user provided data) is known as a bound form instance.

At this point, we still don't know if a user provided valid data with respect to a Django form's field definitions (e.g. if values are text or a valid email). To validate a form's data, you must use the is\_valid() helper method on the bound form instance. If form.is\_valid() is True the data is processed and subsequent action is taken, this additional action consists of redirecting control to the index url. If form.is\_valid() is False it means the form data has errors, after which point control falls to the last return statement which now passes a bound form instance to render the template. By using a bound form instance in this last case, the user gets a rendered form filled with his initial data submission and errors so he's able to correct the data without reintroducing values from scratch.

## **Functional web form syntax for Django forms**

So far you've learned how a Django form class definition can quickly be turned into an HTML form. But this automatic HTML generation is only part of the benefit of using Django form class definitions, you can also validate form values and present errors to end users much more quickly.

To perform these last actions, it's first necessary to have a functional web form. Below illustrates the template syntax to create a functional web form from a Django form.

#### Django form template declaration for functional web form

<form method="POST">

{% csrf\_token %}

{{form.as\_p}}

<input type="submit" value="Submit form">

</form>

The first thing to notice is the form is wrapped around the HTML <form> tag which is standard for all web forms. The reason Django forces you to explicitly set the <form> tag is because its attributes dictate much of a web form's behavior and can vary depending on the purpose of the form. The method attribute tells a web browser that when the form is submitted it POST the data to the server. The POST method value is standard practice in web forms that process user data -- an alternative method option value is GET, but it's not a typical choice for transferring user provided data. The {% csrf\_token %} statement  is a Django tag. {% csrf\_token %} is a special tag reserved for cases when a web form is submitted via POST and processed by Django. The csrf initials mean **Cross-Site Request Forgery,** which is a default security mechanism enforced by Django. While it's possible to disable CSRF and not include the {% csrf\_token %} Django tag in forms, I would advise against it and recommend you keep adding the {% csrf\_token %} Django tag to all forms with POST, as CSRF works as a safeguard and mostly behind the scenes. Next is the {{form.as\_p}} which represents the Django form instance and outputs the HTML output  Finally, there's the <input type="submit"> tag which generates the form's submit button -- that when clicked on by a user submits the form -- and the closing </form> tag.

## **CSRF: What is it and how does it work with Django ?**

**CSRF or Cross-Site Request Forgery** is a technique used by cyber-criminals to force users into executing unwanted actions on a web application. When users interact with web forms, they make all kinds of state-changing tasks that range from making orders (e.g. products, money transfers) to changing their data (e.g. name, email, address). Most users tend to feel a heightened sense of security when they interact with web forms because they see an HTTPS/SSL security symbol or they've used a username/password prior to interacting with a web form, all of which leads to a feeling there's no way a cyber-criminal could eavesdrop, guess or interfere with their actions.

A CSRF attack relies for the most part on social engineering and lax application security on a web application, so the attack vector is open irrespective of other security measures (e.g. HTTPS/SSL, strong password). Below illustrates a CSRF vulnerable scenario on a web application.

[https://www.webforefront.com/static/images/beginningdjango/Figure_6-2.png](https://www.webforefront.com/static/images/beginningdjango/Figure_6-2.png)

#### Web application with no CSRF protection

After user "X" interacts with web application "A" (e.g. making an order, updating his email) he simply navigates away and goes to other sites. Like most web applications, web application "A" maintains valid user sessions for hours or days, in case users come back and decide to do other things without having to sign-in again. Meanwhile, a cyber-criminal has also used site "A" and knows exactly where and how all of its web forms work (e.g. urls, input parameters such as email, credit card).

Next, a cyber-criminal creates links or pages that mimic the submission of web forms on web application "A". For example, this could be a form that changes a user's email in order to overtake an account or transfers money from a user's account to steal funds. The cyber-criminal then seeds the Internet with these links or pages through email, social media or other web sites with enticing or frightening headlines: "Get a $100 coupon from site 'A'", "Urgent: Change your password on site 'A' because of a security risk'. In reality, these links or pages don't do what they advertise, but instead in a single click mimic web form submissions from site "A" (e.g. change a user's email or transfer funds).

Now lets turn our attention to unsuspecting user "X" that visited site "A" hours or days earlier. He catches a glimpse of these last advertisements and thinks "Wow, I can't pass this up". Thinking what harm can a click do, he clicks on the bogus advertisement, the user is then sent to a legitimate site 'A' page as a façade or the click 'appears' to have done nothing. User "X" thinks nothing of it and goes back to some other task. If site 'A' did not have web forms with CSRF protection, then user "X" just inadvertently -- in a single click -- performed an action on site "A" he wasn't aware of.

As you can see, in order to perform a CSRF attack all that's needed is for a user to have an active session on a given site and a cyber-criminal crafty enough to trick a user into clicking on a link or page that performs actions on said site. Hence the term's name: "Cross-Site", because the request doesn't come from the original site, and "Request Forgery" because it's a forged request by a cyber-criminal.

To protect against web form CSRF attacks, it's isn't sufficient for web applications to trust authenticated users, because as I've just described authenticated users could have inadvertently triggered actions they weren't aware of. Web forms must be equipped with a unique identifier -- often called a CSRF token -- that's unique to a user and has an expiration time, similar to a session identifier. In this manner, if a request is made by an authenticated user to a site, only requests that match his CSRF token are considered valid and all other requests are discarded, as illustrated below

[https://www.webforefront.com/static/images/beginningdjango/Figure_6-3.png](https://www.webforefront.com/static/images/beginningdjango/Figure_6-3.png)

As I figure, the inclusion of a CSRF token in a web form makes it very difficult to forge a user's request.

In Django, a CSRF token is generated in web forms with the {% csrf token %} tag that generates an HTML tag in the form <input type="hidden" name="csrfmiddlewaretoken" value="32\_character\_string">, where the 32-character string value varies by user. In this manner, if a Django application makes a POST request -- like those made by web forms -- it will only accept the request if the CSRF token is present and valid for a given user, otherwise it will generate a '403 Forbidden' page error.

**Relative URLs**

So far when we have had to use an anchor tag with an href we have passed in a hardcoded path to the file. This is poor practice if we want our Django project to work on any system or scalable

**The url tag**

The url tag helps us to generate links in the templates. It has the following syntax:

**Syntax:** **{% url 'url\_name' arg1 arg2 %}**

where url\_name is the value we passed to the name keyword argument of the url() function. The arg1 and arg1 are additional arguments required by the view function. On success, it returns part of the URL without host portion. If it can’t create URL NoReverseMatch exception is thrown.

Eg:1,

**Inside proj urls.py file**

urlpatterns=[

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

url(r’^$’, views.index, name=”index”),

url(r’^relative ,views.relative, name=”relative”),]

**views.py**

def index(request):

return render(request,’index.html’)

**index.html**

<!DOCTYPE html>

<html>

<head>

<title>my page1</title>

</head>

<body>

<h1>This is My first page</h1>

</body>

</html>

**Next I add a link to index.html in relative.html**

**relative.html**

<!DOCTYPE html>

<html>

<head>

<title>my page2 </title>

</head>

<body>

**<a href=”{% url ‘index’ %}”>Link to index page</a>**

</body>

</html>

Here we used ‘index’ the name of the view specified in urls.py file.

Suppose the url corresponding to index page specified in an application url rather than project url, reconstruct the link to it using the **{% url 'app\_name:page\_name' %}** template tag with the app's name and the page's name:This method requires that app\_name variable to be created inside the urls.py file. So we have to add inside the urls.py file ,the variable app\_name. Then set this variable equal to a string that has the same name as our app name.

Eg:2,

**Inside app1:url**

**app\_name=app1**

urlpatterns=[

url(r’^$’,views.index,name=”index”),

url(r’^relative,views.relative,name=”relative”),]

**views.py**

def index(request):

return render(request,’index.html’)

**index.html**

<!DOCTYPE html>

<html>

<head>

<title>my page1</title>

</head>

<body>

<h1>This is My first page</h1>

</body>

</html>

Next I add a link to index.html in relative.html

**relative.html**

<!DOCTYPE html>

<html>

<head>

<title>my page2 </title>

</head>

<body>

<a href=”{% url ‘app1:index’ %}”>Link to index page</a>

</body>

</html>

Here we used application name ‘app1’ and ‘index’ the name of the view specified in urls.py file.

**TEMPLATE INHERITANCE/EXTENDING**

So far we’ve created several Django HTML templates for different pages in the application. While most sites will have lots of repeated structure (i.e. headers, sidebars, footers, etc) repeating the HTML in each template is a not good way to handle this. So instead of doing the same cut and paste hack job, we can minimize the amount of repetition in our code base by employing template inheritance provided by Django’s Template Language.

The basic approach to using inheritance in templates is as follows.

1. Identify the re-occurring parts of each page that are repeated across your application (i.e. header bar, sidebar, footer, content pane)

2. In a base template, provide the skeleton structure of a standard page along with any common content (i.e. the copyright notice that goes in the footer, the logo and title that appears in the section), and then define a number of blocks which are subject to change depending on which page the user is viewing.

3. Create specific templates - all of which inherit from the base template - and specify the contents of each block.

Templates extending helps when you want to use the same information or layout in more than one place. You don't have to repeat yourself in every file. And if you want to change something, you don't have to do it in every template, just one!

So first we create a base template. This template is the most basic template that you extend on every page of your website. For extending capability, we need to include template tags in base html file  to indicate what can be overridden in the base template - this is done through the **use of blocks.**

For eg:

Suppose base template as follows:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

**{% block title %}My BASE PAGE{% endblock %}**

<link rel=”stylesheet” href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css>

</head>

<body>

<nav class="navbar navbar-default navbar-static-top">

<div class="container">

<ul class="nav navbar-nav">

<li><a class="navbar-brand" href="{% url 'index’ %}">HOME</a></li>

<li><a class="navbar-link" href="{% url 'app :other' %}">OTHER</a></li>

</ul>

</div>

</nav>

<div class="container">

**{% block body\_block %}**

**{# Anything outside of this will be inherited if you use extend.#}**

**{% endblock %}**

</div>

</body>

</html>

We introduce two new features into the template.

* The first is a new Django template block, title. This will allow us to specify a custom page title for each page inheriting from our base template. If an inheriting page does not make use of this feature, the title is defaulted to MY BASE PAGE.
* We place a navigation bar above our **body\_block** block. This will ensure the navbar is present across all pages inheriting from the base template.

Now that we’ve created a base template with a block, we can now update the templates we have created to inherit from the base template.

To do this, first remove all the repeated HTML code leaving only the HTML and Template Tags/Commands specific to the page. Then at the beginning of the template add the following line of code:

**{% extends 'base.html' %}**

**Index.html**

<!DOCTYPE html>

**{% extends "base.html" %}**

{% block title %}{{ title}}{% endblock %}

{% block body\_block%}

<h1>Hello and welcome to the site!</h1>

<h1>This is the index.html page</h1>

{% endblock %}

Now that we inherit from base.html, all that exists within the index.html template is the extends command, the title block and the body\_block block. You don’t need a well-formatted HTML document because base.html provides all the groundwork for you. All you’re doing is plugging in additional content to the base template to create the complete HTML document which is sent to the client’s browser.

**Views.py:**

def index(request):

context={“title”:”MY INDEX PAGE”}

return render(request,index.html,context)

def other(request):

context={“title”:”MY OTHER PAGE”}

return render(request,other.html,context)

**project urls.py**

urlpatterns=[

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

url(r’^$’,views.index,name=”index”),

url(r’^app/’,include(‘app.urls’)),]

**Application app:urls.py**

app\_name=”app”

urlpatterns=[

url(r’^$’,views.other,name=”other”),]

**Other.html**

<!DOCTYPE html>

{% extends "base.html" %}

{% block title %}{{ title}}{% endblock %}

{% block body\_block %}

<h1>This is an example of template inheritance!</h1>

<h2>Officially this is the other.html page!</h2>

{% endblock %}

This page also extending the content of base template and adds additional content specific to this page inside block tag.