

Preparing environment

<code>mkdir project_name && cd \$_</code>	Create project folder and navigate to it
<code>python -m venv env_name</code>	Create venv for the project
<code>source env_name/bin/activate</code>	Activate environment (Replace "bin" by "Scripts" in Windows)
<code>pip install django</code>	Install Django (and others dependencies if needed)
<code>pip freeze > requirements.txt</code>	Create requirements file
<code>pip install -r requirements.txt</code>	Install all required files based on your pip freeze command
<code>git init</code>	Version control initialisation, be sure to create appropriate gitignore

Create project

<code>django-admin startproject mysite</code> (or I like to call it config)	This will create a mysite directory in your current directory the manage.py file
<code>python manage.py runserver</code>	You can check that everything went fine

Database Setup

<code>OPENUP_MYSITE_SETTINGS.py</code>	It's a normal Python module with module-level variables representing Django settings.
<code>ENGINE = 'django.db.backends.sqlite3', 'django.db.backends.postgresql', 'django.db.backends.mysql', or 'django.db.backends.oracle'</code>	If you wish to use another database, install the appropriate database bindings and change the following keys in the DATABASES 'default' item to match your database connection settings
<code>NAME</code> – The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, NAME should be the full absolute path, including filename, of that file.	The default value, <code>BASE_DIR / 'db.sqlite3'</code> , will store the file in your project directory.
If you are not using SQLite as your database, additional settings such as USER, PASSWORD, and HOST must be added.	For more details, see the reference documentation for DATABASES .

Creating an app

<code>python manage.py startapp app_name</code>	Create an app_name directory and all default file/folder inside
<code>INSTALLED_APPS = ['app_name', ...]</code>	Apps are "pluggable", that will "plug in" the app into the project



Creating an app (cont)

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

Into urls.py from project folder, include app urls to project

Creating models

```
class ModelName(models.Model)
    title = models.CharField(max_length=100)

    def __str__(self):
        return self.title
```

Create your class in the app_name/models.py file

Create your **fields**

It's important to add `__str__()` methods to your models, because objects' representations are used throughout Django's automatically-generated admin.

Database editing

```
python manage.py makemigrations app_name
python manage.py sqlmigrate #identifier
python manage.py check
python manage.py migrate
python manage.py shell
```

By running makemigrations, you're telling Django that you've made some changes to your models

See what SQL that migration would run.

This checks for any problems in your project without making migrations

Create those model tables in your database

Hop into the interactive Python shell and play around with the free API Django gives you

Administration

```
python manage.py createsuperuser
admin.site.register(ModelName)
http://127.0.0.1:8000/admin/
```

Create a user who can login to the admin site

Into app_name/admin.py, add the model to administration site

Open a web browser and go to "/admin/" on your local domain

Management

```
mkdir app_name/management app_name/management/commands &&
cd $_
touch your_command_name.py
```

Create required folders

Create a python file with your command name



Management (cont)

```
from django.conf.urls import include, url
from django.contrib import admin
from django.urls import path, re_path, include
from django.views import generic
from django.views.decorators.csrf import csrf_exempt
from django.views.decorators.http import require_http_methods
```

Edit your new python file, start with import

```
class Command(BaseCommand):
```

Create the Command class that will handle your command

```
    help = "This message will be shown with the --help option after your command"
```

```
    def handle(self, args, **kwargs):
```

```
        # Work the command is supposed to do
```

```
python manage.py my_custom_command
```

And this is how you execute your custom command

Django lets you create your custom CLI commands

Write your first view

```
from django.http import HttpResponse
def index(request):
    return HttpResponse("Hello, world. You're at the index.")
```

Open the file app_name/views.py and put the following Python code in it.

This is the simplest view possible.

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

In the app_name/urls.py file include the following code.

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

View with argument

```
def detail(request, question_id):
    return HttpResponse(f"You're looking at question {question_id}")
```

Example of view with an argument

```
urlpatterns = [
    path('<int:question_id>/', views.detail, name='detail'),
    ...
```

See how we pass argument in path

```
{% url 'app_name:view_name' question_id %}
```

We can pass attribute from template this way



By Olivier R. (OGR)
cheatography.com/ogr/

Published 6th February, 2022.
Last updated 12th February, 2022.
Page 3 of 8.

Sponsored by [Readable.com](https://readable.com)
Measure your website readability!
<https://readable.com>

View with Template

<code>app_name/ templates /app_name /index.html</code>	This is the folder path to follow for template
<code>context = {'key': value}</code>	Pass values from view to template
<code>return render (request, 'app_name /index.html', context)</code>	Exemple of use of render shortcut
<code>{% Code %}</code>	Edit template with those. Full list here
<code>{{ Variable from view's context dict }}</code>	
<code></code>	
<code><title>Page Title</title></code>	you can put this on top of your html template to define page title

Add some static files

<code>'django.contrib.staticfiles'</code>	Be sure to have this in your INSTALLED_APPS
<code>STATIC_URL = 'static/'</code>	The given examples are for this config
<code>mkdir app_name/ static app_name/ static /app_name</code>	Create static folder associated with your app
<code>{% load static %}</code>	Put this on top of your template
<code><link rel="stylesheet" type="text/css" href="{% static 'app_name /style.css' %}" ></code>	Exemple of use of static.

Raising 404

<code>raise Http404("Question does not exist")</code>	in a try / except statement
<code>question = get_object_or_404(Question, pk=question_id)</code>	A shortcut

Forms

<code>app_name/ forms.py</code>	Create your form classes here
<code>from django import forms</code>	Import django's forms module
<code>from .models import YourModel</code>	import models you need to work with
<code>class ExempleForm(forms.Form): exemple_field = forms.CharField(label='Exemple label', max_length=100)</code>	For very simple forms, we can use simple Form class
<code>class ExempleForm(forms.ModelForm): class meta: model = model_name fields = ["fields"] labels = {"text": "label_text"} widget = {"text": forms.Widget_name}</code>	A ModelForm maps a model class's fields to HTML form <input> elements via a Form. Widget is optional. Use it to override default widget
<code>TextInput, EmailInput, PasswordInput, DateInput, Textarea</code>	Most common widget list
<code>if request.method != "POST": form = ExempleForm()</code>	Create a blank form if no data submitted



Forms (cont)

```
form = ExampleForm(data=request.POST)
```

The form object contains the information submitted by the user

```
if form.isvalid():
    form.save()
    return redirect("app_name:view_name", argument=argument)

{% csrf_token %}
```

Form validation. Always use redirect function

Template tag to prevent "cross-site request forgery" attack

Render Form In Template

```
{{ form.as_p }}
```

The most simple way to render the form, but usually it's ugly

```
{{ field|placeholder:"field_label" }}
{{ form.user_name|placeholder:"Your name here" }}
```

The | is a filter, and here for placeholder, it's a custom one. See next section to see how to create it

```
{% for field in form %}
{{form.username}}
```

You can extract each field with a for loop. Or by explicitly specifying the field

Custom template tags and filters

```
app_name \templates \__init__.py
```

Create this folder and this file. Leave it blank

```
app_name \templates \filter_name.py
```

Create a python file with the name of the filter

```
{% load filter_name %}
```

Add this on top of your template

```
from django import template
```

To be a valid tag library, the module must contain a module-level variable named register that is a template.Library instance

```
register = template.Library()
```

```
@register.filter(name='cut')
```

Here is an example of filter definition.

```
def cut(value, arg):
```

See the decorator? It registers your filter with your Library instance.

```
    """ Removes all values of arg from the given string """
    return value.replace(arg, '')
```

You need to restart server for this to take effects

<https://tech.serhatteker.com/post/2021-06/placeholder-templatetags/>

Here is a link of how to make a placeholder custom template tag



Setting Up User Accounts

Create a "users" app

```
app_name = "users"

urlpatterns[
    # include default auth urls.
    path("", include("django.contrib.auth.urls"))
]
```

Don't forget to add app to settings.py and in
from users.

Inside app_name/urls.py (create it if inexistent)
this code includes some default authentication
Django has defined.

```
{% if form.error %}
    <p>Your username and password didn't match</p>
{% endif %}
<form method="post" action="{% url 'users :login' %}">
    {% csrf_token %}
    {{ form.as_p }}

    <button name="submit">Log in</button>
    <input type="hidden" name="next" value="{% url 'app_name :index' %}" />
</form>
```

Basic login.html template

Save it at save template as

users/templates/registration/login.html

We can access to it by using

```
<a href="{% url 'users :login' %}">
a>
```

```
{% if user.is_authenticated %}
```

Check if user is logged in

```
{% url "users: logout" %}
```

Link to logout page, and log out the user
save template as users/templates/registration/registered.html

```
path("register/", views.register, name="register"),
```

Inside app_name/urls.py, add path to register

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth import UserCreationForm
```

We write our own register() view inside use
For that we use UserCreationForm, a Django
model.

```
def register(request):
    if request.method != "POST":
        form = UserCreationForm()
    else:
        form = UserCreationForm(data=request.POST)

    if form.is_valid():
        new_user = form.save()
        login(request, new_user)
        return redirect("app_name:index")

    context = {"form": form}
    return render(request, "registration/register.html", context)
```

If method is not post, we render a blank form
Else, is the form pass the validity check, and
We just have to create a registration.html template
folder as the login and logged_out

Allow Users to Own Their Data

```
...
from django.contrib.auth.decorators import login_required
...

@login_required
def my_view(request):
    ...

...
from django.contrib.auth.models import User
...
owner = models.ForeignKey(User, on_delete=models.CASCADE)

user_data = ExampleModel.objects.filter(owner=request.user)

...
from django.http import Http404
...

...
if example_data.owner != request.user:
    raise Http404

new_data = form.save(commit=False)
new_data.owner = request.user
new_data.save()
```

Restrict access with `@login_required` decorator

If user is not logged in, they will be redirected to the login page
To make this work, you need to modify `settings.py` so Django knows where to find the login page
Add the following at the very end

```
# My settings
LOGIN_URL = "/login/"
```

Add this field to your models to connect data to certain users

When migrating, you will be prompted to select a default value

Use this kind of code in your views to filter data of a specific user

`request.user` only exist when user is logged in

Make sure the data belongs to the current user

If not the case, we raise a 404

Don't forget to associate user to your data in corresponding views

The "commit=False" attribute let us do that

Paginator

```
from django.core.paginator import Paginator

example_list = Example.objects.all()

paginator = Paginator(example_list, 5) # Show 5 items per page.

page_number = request.GET.get('page')

page_obj = paginator.get_page(page_number)

{% for item in page_obj %}
```

In `app_name/views.py`, import `Paginator`

In your class view, Get a list of data

Set appropriate pagination

Get actual page number

Create your Page Object, and put it in the context

The Page Object acts now like your list of data



Paginator (cont)

```
<div class="pagination">
  <span class="step-links">
    {% if page_obj.has_previous %}
      <a href="?page=1">&laquo; first</a>
      <a href="?page={{ page_obj.previous_page_number }}">previous</a>
    {% endif %}
    <span class="current"> Page {{ page_obj.number }} of {{ page_obj.paginator.num_pages }}. </span>
    {% if page_obj.has_next %}
      <a href="?page={{ page_obj.next_page_number }}">next</a>
      <a href="?page={{ page_obj.paginator.num_pages }}">last &rarr;</a>
    {% endif %}
  </span>
</div>
```

An
exemp
of wha
to put
the
bottom
of your
page
to
naviga
through
Page
Object

Deploy to Heroku

<https://heroku.com>

Make a Heroku account

<https://devcenter.heroku.com/articles/heroku-cli/>

Install Heroku CLI

```
pip install psychog2
pip install django -heroku
pip install gunicorn
```

install these packages

```
pip freeze > requirements.txt
```

update requirements.txt

```
# Heroku settings.
import django_heroku
django_heroku.settings(locals(), staticfiles= False)
if os.getenv('DEBUG') == "TRUE":
    DEBUG = True
elif os.getenv('DEBUG') == "FALSE":
    DEBUG = False
```

At the very end of settings.py, make an Heroku settings section
import django_heroku and tell django to apply django heroku settings
The staticfiles to false is not a viable option in production, check
whitenoise for that IMO



By Olivier R. (OGR)
cheatography.com/ogr/

Published 6th February, 2022.
Last updated 12th February, 2022.
Page 8 of 8.

Sponsored by [Readable.com](https://readable.com)
Measure your website readability!
<https://readable.com>