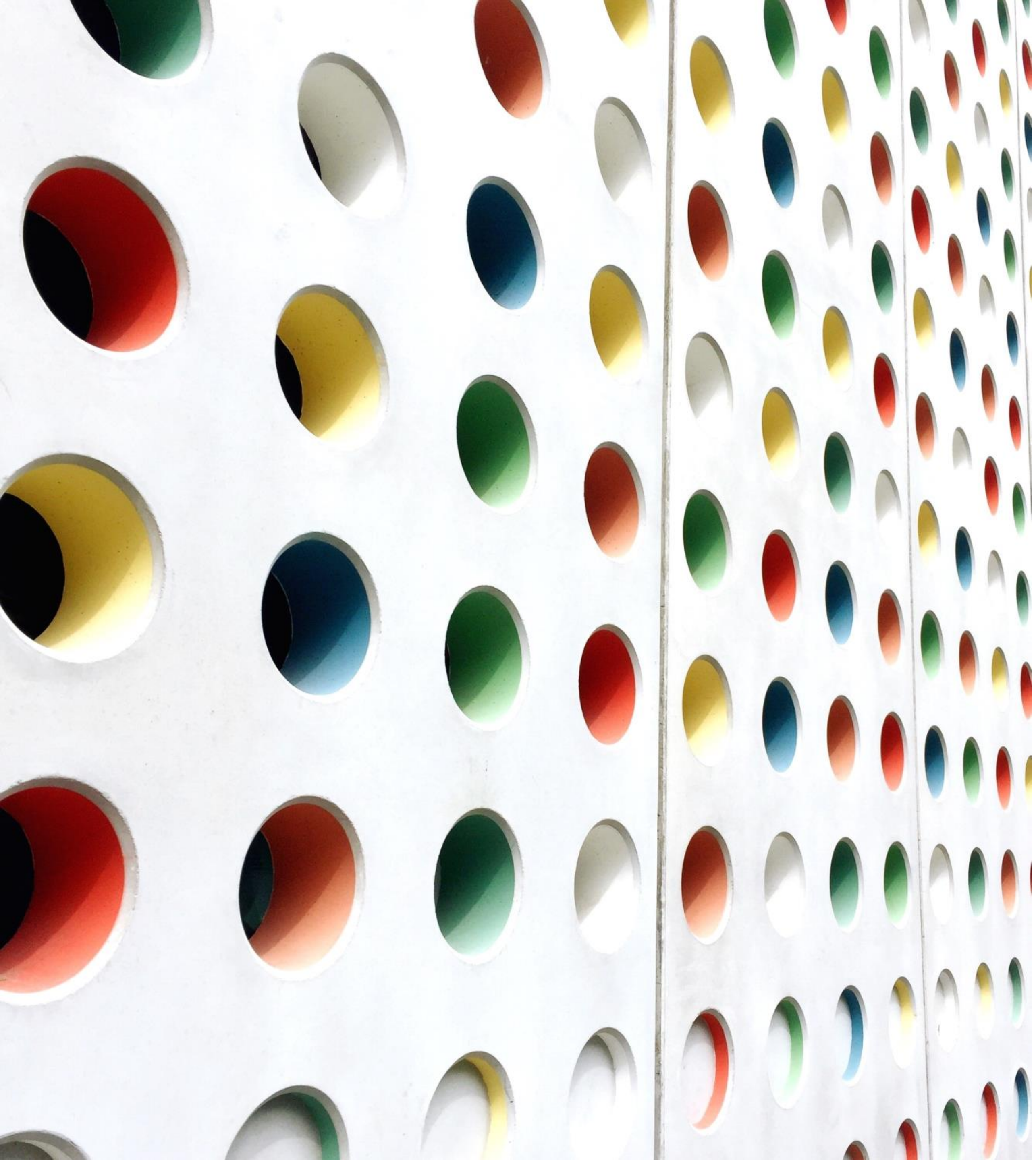


Django Project

DJANGOPROJECT.COM BY NEIL, MAIHESUTI, & GREGORY

What is a Django

DJANGO + PYTHON = FAST

What is a Django

Django is a popular web framework built with Python that allows developers to quickly create maintainable web applications. Django as a Python web framework that allows you to rapidly develop web projects in a clean, pragmatic design. A web framework is a collection of tools used to build web applications and websites. Django comes with tools like an object-relational mapper, or ORM, which helps us make database queries. It also offers URL routing which determines what logic to follow depending on the URL of a web request. HTML templating is another feature of Django which facilitates viewing the data as HTML in a web browser. Django itself is not a programming language. Python is the programming language that Django was built with. Django is also not a web server but it does have a built-in web server for local development.

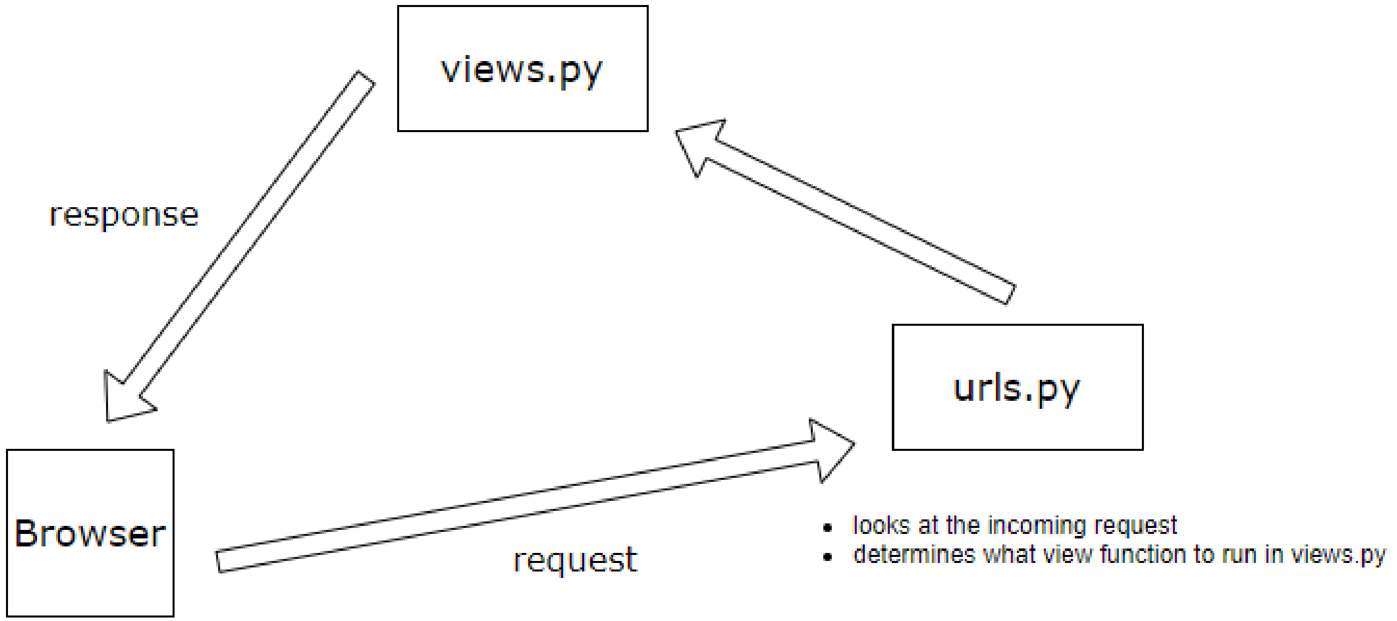
Create New Project

With Python and Django already installed on our system, all we need to do is to navigate to a directory of our choice and enter the command

django-admin startproject djangoblog

In less than a minute we have a Django project!

Diagram on the right is showing how URLs and Views work together in Django


views.py
response
urls.py
Browser
request
looks at the incoming request
determines what view function to run in views.py

Django and HTML pages

Django is written in Python. Django and Python work a little differently. In Django, DTL templates to output dynamic data generated by Python and Django into HTML pages.

A `templates` directory must be created in the project root folder. In our case, the project root is the outer d*jangoblog* directory.

HTML templates render output in browser instead of sending an HttpResponse directly from view functions.

Adding an App

After creating djangoblog project we added a new app to our top-level project folder using the python manage.py startapp command

```
python manage.py startapp posts
```

This app is to manage posts in a blog system.

Register the App

Every time we add new app to a project, it must be registered in the settings.py file

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

Including URLs

urlpatterns must be defined inside of a Django application, to be included in the main urls.py

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

urlpatterns = [
    path('', views.home),
    path('admin/', admin.site.urls),
    path('posts/', include('posts.urls')),
    path('about/', views.about)
]
```

A Post Model

We are using a model concept of loading dynamic data from a database to the website. A blog application stores posts in the database. Models.py files in Django are a class which represents a table in a database.

When we ran

```
python manage.py startapp posts command
```

Django created models.py file so we can define one or more model classes

Define a Model

The first step for working with a Model in Django is to create the Model class in the models.py file. To define database table we created a Python Class in Models.py file

```
from django.db import models
# Create your models here.
class Post(models.Model):
    title = models.CharField(max_length=100)
    slug = models.SlugField()
    body = models.TextField()
    date = models.DateTimeField(auto_now_add=True)
```

Django Migrations

Migrations is the step of mapping the Model class to a table in the database, every time we make or update to a model, we need to also update the migration by running commands

```
python manage.py makemigrations
```

```
python manage.py migrate
```

The Django ORM

The object-relational mapper bridges between the code and the database. The ORM provides full create, read, update, and delete functionality for the Model we have defined.

The Python Class Model is now mapped to a table in the database, any time we make a new model or change an existing model, we need to run python

manage.py makemigrations

and then we need to run

python manage.py migrate.

Admin. Create a superuser

In order to add admin functionality we needed to add a superuser with the manage.py file with the command python manage.py

```
python manage.py createsuperuser
```

```
Username (leave blank to use 'myusername'): admin
Email address: admin@example.com
Password:
Password (again):
```

Once a superuser created, type http://127.0.0.1:8000/admin address in a web browser to login to an admin area

Static Files and Images

In our blog we wanted to use images. Django allow to locate static files and images. In Django, we can have static files shared across an entire project or on a per-application basis. We needed to configure the STATICFILES_DIRS constant in settings.py.

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static’)
```

Later to be used in HTML files

```
<img src="{% static 'posts/images/'|add:post.slug|add:'.jpeg' %}" class="card-img-top">
```

Slug Definition

Defining URL path for slug capture so when web applications respond to Http requests and use parameters in the request to determine how to route, we can visit links http://localhost:8000/posts/a-new-title/, http://localhost:8000/posts/second-post/, and http://localhost:8000/posts/third-post/ and Django will route to each particular Post detail view. To capture a slug variable in the URL, we use the <slug:slug> patten in the path() function

This is shown the code in posts/urls.py file

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.post_list),
    path('<slug:slug>/', views.post_detail, name='post_detail'),
]
```

Sign Up and Log In Users

Django has some tools built in that you can use to set up the ability for users to log in and log out of a web application.

`UserCreationForm` and the `AuthenticationForm` classes provide a way to scaffold out the entire sign up and login configuration of the forms to handle these tasks, with validation built right in. We created a new app in our Django project named `accounts` for login and log out functions.

```
python manage.py startapp accounts
```

then we need to update djangoblog/settings.py file in INSTALLED_APPS sections to add 'accounts'

Including URLs

The highlighted code in main djangoblog/urls.py ensures that anytime a user visits http://example.com/accounts, they will then be making use of the url patterns defined

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

urlpatterns = [
    path('', views.home),
    path('admin/', admin.site.urls),
    path('accounts/', include('accounts.urls')),
    path('posts/', include('posts.urls')),
    path('about/', views.about)

]
```

Define View Function

In accounts directory we defined a signup_view() in `accounts/views.py` file which accepts the Http request, then returns a render of the signup.html template

```
from django.shortcuts import render

def signup_view(request):

    return render(request, 'accounts/signup.html')
```

HTML templates: Sign Up

We added a new directory and provide the path of templates/accounts for our templates in accounts/templates/accounts, and created signup.html file

```
{% extends "./layout/base.html" %}
{% load static %}

{% block content %}
    <h1>Signup</h1>
    <form action="/accounts/signup/" method="post">
        {{ form }}
    </form>
{% endblock content %}
```

HTML templates: Base

We added a accounts/templates/accounts/layout/base.html file

```
<!doctype html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    {% load static %}
    <link rel="stylesheet" href="{% static "posts/css/bootstrap.min.css" %}">
    <title>Django Blog</title>
</head>
<body class="container mt-5">
{% block content %}
{% endblock content %}

</body></html>
```

HTML templates: Login

We included `{% csrf_token %}` for validation to ensures that the request is valid and secure.

```
{% extends "./layout/base.html" %}
{% load static %}
{% block content %}
    <h1>Log In</h1>
    <form action="{% url 'accounts:login' %}" method="post">
        {% csrf_token %}
        {{ form }}
        <input type="submit" value="Log In">
    </form>
{% endblock content %}
```

Log User Into Application

In accounts/view.py added the code to log in a user after form submitted and redirected, to establish a session

```
from django.contrib.auth import login
def signup_view(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
            return redirect('posts:list')
    else:
        form = UserCreationForm()
    return render(request, 'accounts/signup.html', {'form': form})
```

Log User Into Application (continue)

```
def login_view(request):
    if request.method == 'POST':
        form = AuthenticationForm(data=request.POST)
        if form.is_valid():
            user = form.get_user()
            login(request, user)
            return redirect('posts:list')
    else:
        form = AuthenticationForm()
    return render(request, 'accounts/login.html', {'form': form})
```

Logout

In posts/templates/posts/layout/base.html we added is_authenticated property on the user object {% if user.is_authenticated %}. It display a log out button to the user, but only if the user is currently logged in.

```
<nav class="mb-3">
    {% if user.is_authenticated %}
        <form action="{% url 'accounts:logout' %}" method="post">
            {% csrf_token %}
            <button class="btn btn-primary btn-sm" type="submit">Log Out</button>
        </form>
    {% endif %}
</nav>
```

References

https://vegibit.com/how-to-sign-up-and-log-in-users-with-django/

https://djangocentral.com/building-a-blog-application-with-django

https://www.djangoproject.com/

https://docs.djangoproject.com/en/3.1/topics/install