First Step

To create a django project, we use the django CLI.

```
$ django-admin start project <Project-Name>
```

this command will create the default django files needed

Let's learn the list of mainly generated django files

1. `manage.py` is the main file which we don't change anything about just now
2. `__init__.py`
3. `asgi.py` and `wsgi.py` these files are needed when you want to deploy and host a django project later on.
4. `settings.py` the name tell all, we need to get familiar with this file deeply along the way.
5. `urls.py` all the urls! (routes in flask)

Finally we'll learn about localhost and debug mode.

To start the django webserver on locanhost (debug mode)

```
$ python manage.py runserver
```

And to stop the server from running you can kill the process or terminal using `Ctrl+c`.

Apps

Each app (also known as Module) is a sub program in django which will be used to code a specific part of the entire serverside program.

To create an app:

```
$ python manage.py startapp <App-Name>
```

This will create a folder in the project directory called App-Name, in which you'll fine a set of similar files to the main project folder. I have already create a sample app called app_1, to which I will refere from this point on.

The project directory includes folders for each app. You'll notice there is also a folder named the same as the project directory. That folder is as well a django app which is in charge of the entire project, more like a team-lead!

What new things are inside the app_1 folder

1. `migrations`: we'll talk about this later.
2. `admin.py`: later!
3. `test.py`: Used for unit testing, not talked about in this course.
4. `views.py`: This is the stuff, the file which is in charge of what data is sent to client-side.

You need to introduce each new app you make to django!

To do this, you'll nee to add the app name to the INSTALLED_APPS list in settings.py in the main project app directory.

Nerd out: Search about INSTALLED_APPS and what it does.

URL: Uniform Resource Locator

You know what a URL is. but how to make a URL in django? let's learn that along with views!

Views

View is a logic that is excecuted when each URL is called. A set of functions and classes!

To create a view, make an app first (and don't forget to add it in `settings.py`) and then navigate to `views.py` file. (*Nerd out* about the import line). Add this line to the file:

```
from django.http import HttpResponse

def index(request): # or any name you want
    return HttpResponse("Hello, World!")
```

`request` parameter is what the client side sends to the URL `/` which django will recieve and feed into `index()` function.

Now it's time to introduce the url to the app and therefore the django himslef! make a `urls.py` file in the app directory and add this code inside it.

```
from django.urls import path
from . import views # the views file in the same app directory.

urlpatterns = [
    path('sunday', views.index) # the name of the function you made in
views.
]
```

Unfortunately the work isn't done yet, since we used an app (and not the main app) to make a view and a url, we need to add this url to the main app `urls.py` too! Navigate to main app directory and open `urls.py`. update `urlpatterns` to include the newly made url.

```
from django.urls import path, include # include helps add all urls
from an external app to the main app with a single line of code.
from django.contrib import admin

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

In the line `path('url_test', include('url_test.urls'))`, this first `url_test/` maked a new url like: `localhost:8000/url_test/`. And the second one referes to what you called your external app, they CAN be diffenet indeed!

You can also add a path like this: `path('', include('url_test.urls'))` which directly bonds root url to external app urls.

It is pretty obvious that you need to add a root URL for the main domain. to do that, you must make a `views.py` file in the main app directory and make the desired function for the main page! and then update main app urls to this:

```
from . import views

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

Conclusion

Basically, the URL system in django is more like a tree. The root of the tree lies in the main app directory `urls.py`. Main branches of this tree would be each external app. The leaves so to speak would be the Views!

POST & GET

Before we do anything, let's review a checklist of how to make a view and make it available on a certain url.

1. Make an app (e.g. user) and add the app to `settings.py`
2. Navigate to `views.py` and make any view
3. Make a `urls.py` file in the user directory and add the path to `urlpatterns`
4. Add the path to main app `urlpatterns` as well

GET

Get is essentially a method for sending data from client side to server side. If that data was sent through a URL, we call it a GET request. Note that GET sends data in an exposed URL like this: `https://www.google.com/search?q=Hello`

Therefore GET is not suitable for sensitive data like passwords.

POST

Let's explain in some bullet points by ChatGPT:

- The data is sent in the body of the HTTP request, not in the URL.
- Example: Form submissions, uploading files.
- The request is not idempotent (sending the same POST request multiple times may create duplicate resources or change server state).
- Not cached by default and not visible in browser history.
- No practical limit on the amount of data sent.

Dynamic Addressing

Sometimes it's not very clear how many URLs you need to define for your project, that's where Dynamic Addressing comes in handy. By defining a function that handles all of the needed URLs which might be generated later on.

Let's begin

1. Make a view function as we did before.

```
from django.http import HttpResponse

def dynamic_view(request, dynamic):
    return HttpResponse(dynamic)
```

1. Bind the URL to a path DYNAMICALLY, as follows.

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

urlpatterns = [
    path('<dynamic>', view=views.dynamic_view),
]
```

- The `dynamic` argument will be fed into the `dynamic_view()` method with the same name. (`dynamic`)
- Now, anything the client types in the URL, if is not already binded to another view, will bond to this dynamic view.

1. Test

Type any URL and see what happens: `localhost:8000/MyOwnURL`

Multiple Dynamic URLs

The good thing about Dynamic Addressing is that you can have as many Segments as you need. you just need to update your files in this way:

```
from django.http import HttpResponse

def dynamic_view(request, dynamic, dynamic2, dynamic3):
    return HttpResponse((dynamic, dynamic2, dynamic3))

from django.urls import path
from . import views

urlpatterns = [
    path('<dynamic>/<dynamic2>/<dynamic3>', view=views.dynamic_view),
]
```

Type Annotation for Dynamic Addressing

There also this feature to control which data type the client can put in the URL. This feature is most useful when you want to control which function must be called based on Data Type.

```
urlpatterns = [
    path('<int:dynamic>', view=views.another_view),
    path('<str:dynamic>', view=views.dynamic_view),
]
```

Note that you must put less general data types first.

Because django tries to cast the URL segment to other data types along with some exception handling. Anything can be converted to string but that doesn't check out for int.

HttpResponseRedirect

Among all this Dynamic Addressing crap, one cool thing is Redirecting. This is how you can redirect the client to URL B if URL A was called. **Note that the status code for Redirection will be in range 300.**

```
from django.http import HttpResponseRedirect

def my_view(request):
    return HttpResponseRedirect("path/to/url/B")
```

URL names

Let's learn some best practice about URLs= and redirecting. In a relatively large project, there will be a bunch of URLs, many of them redirecting to each other with something like above.

If you were to change some tiny part of a URL (I mean the path/to/url part), there will be CONSEQUENSES!

To resolve this like a good programmer, Django has the feature of naming urls in `path()` method. You can simply add a keyword argument `name="unique_name"` and then, by using `reverse()` method in the view in which you used a redirection, you can read the URL dynamically.

```
urlpatterns = [
    path('<int:dynamic>', view=views.another_view),
    path('<str:dynamic>', view=views.dynamic_view,
name="unique_name"),
]

from django.urls import reverse

def my_view(request):
    return HttpResponseRedirect(reverse("unique_name", args=[]))
```

`args` is the list of all segments that must appear after the url to which `"unique_name"` is refrencing.

By using this feature, you are sort of storing your URL inside a variable (so to speak) that will look for anytime the URL is called. Will this affect the websites performance? Of course not! it's just like calling a variable by reference.

HTML Response

You must be very well familiar with HTML at this point (go get familiar if you aren't). Of course we're learning Django so we'll be able to send HTML pages back to client side, not plain text!

Let's jump right in. Create a view `index` and bind a URL to it.

```
from django.http import HttpResponse

def index(request):
    response = f"<h1>Hello, HTML!</h1>"
    return HttpResponse(response)
```

Very cool, now let's go a little bit deeper. Let's make a list of things the user can choose from. More like a menu.

```
from django.urls import reverse

def menu(request):
    dest = reverse("root", args=[])
    response = f"""
        <ul>
            <li> <a href="{dest}">Click Me!</a> </li>
        </ul>
    """
    # Of course you can use the entire Python superpower to make these
strings

    return HttpResponse(response)
```

Of course you can use the entire Python superpower to make these strings. However, we will learn better ways of sending HTML docs to client side. By using `render()` method provided in `django.shortcuts`.

Thanks for your attention so far. From this point, we will dive into Templates which is by itself one hell of a topic.

See you later!

Templates

Welcome to a pivot lesson of Django! So far we have been using a string to send responses to client side. Now we will learn the real stuff. Templates are basically folders containing HTML files that can be connected to views and bonded to URLs.

First off, let's take a look at Django project hierarchy once more. (for this project!)

```
MyProject/
    MyProject/
        <files.py>
    app1/
        <files.py>
    app2/
        <files.py>
```

if you want to have a template for each app, you must make a `templates` folder inside the directory of that app. We will make templates for `app1` and `app2` here. Oh! you also need to add a folder with the same name as the app inside the templates folder. Then, you'll be able to add your HTML files.

```
/MyProject
    /MyProject
        <files.py>

    /app1
        /templates
            /app1
                index.html
        <files.py>

    /app2
        /templates
            /app2
                index.html
        <files.py>
```

Let's add some HTML content shall we? (to `/MyProject/app1/templates/app1/index.html`)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
```

```
    <title>App 1</title>
</head>
<body>
    <h1>Hello, Templates!</h1>
    <h2>this is app2</h2>
</body>
</html>
```

And as usual, let's make a view. But this time, we will use a new method called `render_to_string()` which get's the path to `index.html` file and renders it to a string! This method is provided by `django.template.loader`.

p.s. you remember that to make a view you need you alter `/MyProject/app1/views.py`!

```
from django.template.loader import render_to_string
from django.http import HttpResponse

def index(rquest):
    response = render_to_string("app1/index.html")
    return HttpResponse(response)
```

Finally, let's bind it to a URL and give it a test.

```
# /MyProject/app1/urls.py
from django.urls import path
from . import views

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

# /MyProject/MyProject/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('app1/', include("app1.urls"))
]
```

settings.py

We have already mentioned that whenever a new app is made, it must be added to `INSTALLED_APPS` list in `settings.py`. But we never really said why!?

Well if you remove that now, you'll see a couple of errors mentioning that Templates canot be found. The best-practice here is basically adding the app to `settings.py` and ask no more question (for now!). There is however, another way of introducing Templates to Django without adding the app in project setitngs.

Let's take a look at `TEMPLATES` in project settings.

```
from .MyProject.MyProject.settings import *

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

In this python `list[JSON]` called `TEMPLATES` you can add the directory to your app templade (e. g. `BASE_DIR/"app1"/"templates"`) at the `DIRS` key.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR/"app1"/"templates"], # BASE_DIR is defined
in settings.py
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Why add to INSTALLED_APPS?

With all that said, it's not so clear why we added the app to INSTALLED_APPS earlier. Well for starters, it's way easier to add a single word to a list instead of an entire path to a JOSN. But that's really not the reason.

By looking at the key `APP_DIRS` in the TEMPLATES we'll see it is set to `True`. That means that Django is initially configured to look for templated inside anything that is considered an `APP`, and anything that is added to INSTALLED_APPS is indeed and `APP`!

Nevertheless, there is more to INSTALLED_APPS than we mentioned, we'll learn about those in Database lessons later on.

Render Method

At the beginning of our journey, we witnessed a `render()` method imported to `views.py` file from `django.shortcuts`. Which we mentioned (or maybe not) that this is way cooler to use than `HttpResponse`.

What is does, is very basic. If you take a look at the behind-the-scene codes for `render()`, you'll see this.

```
from django.template import loader

def render(
    request, template_name, context=None, content_type=None,
status=None, using=None
):
    """
    Return an HttpResponse whose content is filled with the result of
calling
    django.template.loader.render_to_string() with the passed
arguments.
    """
    content = loader.render_to_string(template_name, context, request,
using=using)
    return HttpResponse(content, content_type, status)
```

You can that `render()` isn't doing anything so magical, yet gives us exactly what we need!

To use it ourselves we only need to return something like `render(request, 'path/to/file.html')` and Voalla!

```
#/MyProject/app1/views.py
def r_index(request):
    return render(request, "app1/index.html")
```

One significant point is that what we're rendering and sending back to client side is infact static! Meaning that we can't change the data in that HTML file based on the client side's request. However, if we take another look at the render behind-the-scene codes, we notice some parameter called `context`, good news, that solves our problem!

`context` is essentially a dictionary that can be rendered alongside our HTML file.

```
#/MyProject/app1/views.py
def r_index(request):
    data: dict = {
        'name':'Erfan',
        'family':'Rajati'
```

```
    }

    return render(request, "app1/render_index.html", context=data)
```

Now, how to use the data in that dictionary in our HTML file? By using the `{{ python_syntax }}` notation inside any tag you desire.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>App 1</title>
</head>
<body>
    <h1>Hello, Templates!</h1>
    <h2>this page is using render method.</h2>

    <h3>
        name: {{ name }}
    </h3>
    <h3>
        lastname: {{ family }}
    </h3>
</body>
</html>
```

This special syntax is called **DTL, Django Template Language.** Cool thing is that you can use a lot of Python Syntax in between two curly braces, There is things beyond Python syntax too! DTL is by itself an entire language based on Python.

Django Template Filters (By DTL)

There is a lot of features made available for django developers which is called Django Template Filters and Django Template Tags. Below, you'll find some examples of how you can use these filters in your HTML file.

```
{{ value|add:'2' }}
{{ first|add:second }}
```

If you're interested, visit this link to see more.

With that said, it's not like the end of the world if you don't learn DTF. You can handle all modifications in `.py` files on your own. At the next chapter, we'll take a look at Django Template Tags, which are, unlike filters, quite important to know!

Django Template Tags (By DTL)

Let's say template tags are talking about the correct syntax that is to be used from Python inside HTML. You can't just put any Python sh*t you know between two curly braces and get away with it! There is significant structure to be used in there.

Let's start simple, just like the project at chapter 7, we have and app called "app1" That has a view which is meant to send some Python list items back to client side inside an HTML file. Let's take a look at the codes.

```
#/MyProject/app1/views.py
from django.shortcuts import render

def index(request):
    data = {
        "HoHoHo": ["Santa", "Christmas", "Candy"] # Yes my child it's
Christmas at the time I'm writing this!
    }
    return render(request, "app1/index.html", context=data)
```

```
<!-- /MyProject/app1/templates/app1/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Django Template Tags</title>
</head>
<body>
    <ul>
        {% for word in  HoHoHo %}
        <li>
            {{ word }}
        </li>
        {% endfor %}
    </ul>
</body>
</html>
```

Add the files and give it a test! we'll learn more deeply about DTT from this point on.

URLs in DTT

You remember the unique name we could give to specific urls like this: `path('url', views.index, name="unique_name")`? well we can use that unique name in HTML files too!

```
<body>
    <a href="{% url 'unique_name' %}">
        Click Me!
    </a>
</body>
</html>
```

if statement in DTT

Obvious by the name, you can use conditional statements inside HTML with DTT. The example works with the case where given data is `None`.

```
<body>
    {% if data is not None %}
    <p>{{ data }}</p>
    {% else %}
    <p>Data not available!</p>
    {% endif %}
</body>
</html>
```

Master Template

The idea of having separate templates for each django app is good but far from completely functional. That's because there are a ton of alike parts in a webpage that can be reused instead of re-implemented!

You must already be familiar with the concept of Abstraction and Inheritance in OOP, good news is that we have something faily similar to those when it comes to django templates.

You can have a set of HTML docs as the Master Template of your project. To do this, we only need to make a folder called `templates` at the project root. The hierarchy would look like this after the work is done.

```
/MyProject
    /MyProject
        <files.py>

    /app1
        /templates
            /app1
                index.html
        <files.py>

    /app2
        /templates
            /app2
                index.html
        <files.py>

    /templates                  # Master Template!
        master.html
```

You'll notice the file called `master.html` in the `templates` directory that includes anything your website has among different pages (most of the pages). Like the intersection for most of html files in the project hierarchy. Not only that, you must also add a Django Template Tag called `block` in any place that might be overrided in project apps. Take a look:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>
        {% block page_title %} <!--page_title is like a variable name
for this block-->
            Default Page Title
```

```
        {% endblock page_title %}
    </title>
</head>
    <body>
        {% block content %}

        {% endblock content %}
    </body>
</html>
```

Make sure you have `BASE_DIR / "templates` added to `TEMPLATES` in Project Settings.

```
from MyProject.MyProject.settings import *

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            BASE_DIR / "templates",
        ],
        'APP_DIRS': True,

        # The rest of the dictionary.
    },
]
```

When master template is set up, it's time to move to the `/MyProject/app1/templates/app1/index.html` or any other HTML file you desire.`

```
{% extends "master.html" %}
<!--
    The file will automatically find master.html
    if the master template directory is added to project
-->

{% block page_title %} App1 {% endblock page_title %}

{% block content %}
    <h1>Extends Master Template!</h1>
{% endblock content %}
```

Includes!

You can store a slice of a HTML file to later reuse it in other parts of your project. By making a new folder in `/MyProject/app1/templates/app1/includes` in which you'll be having html files like this:

```
<!--/MyProject/app1/templates/app1/includes/header.html-->
<header>
    <nav>
        <a href="/">click me!</a>
    </nav>
</header>
```

You are now able to stick this slice to any part of the app1 pages using a Django Template Tag called `{% includ %}`

```
<!--/MyProject/app1/templates/app1/index.html-->
{% extends "master.html" %}
<!--
    The file will automatically find master.html
    if the master template directory is added to project
-->

{% block page_title %} App1 {% endblock page_title %}

<!-- ----- -->

{% block content %}
    <h1>Extends Master Template!</h1>
    {% include "app1/includes/header.html" %}
{% endblock content %}
```

The include block supports sending data back to the reference HTML files, something like the `context` parameter that was sent to rendered HTML when using `render()` method.

```
    {% include "app1/includes/header.html" with context="Sent Data"%}
```

And back in the refrence HTML file `header.html`:

```
<header>
    <nav>
        <h3>{{ context }}</h3>
        <a href="/">click me!</a>
    </nav>
</header>
```

Note: The syntax of Django Template Language is not highlited here, so it maked it pretty difficult to understand which words are keywords and whichones are values. So below you will file a list of so-far-used keywords in DTL. Nth beyond this list is a Keyword in DTL examples used so far.

- for / in
- if / elif / else
- block

- include / with
- endfor / endif / endblock ...

404 Not Found!

The not found page is one of the most common parts where you need to work with master templates. Look at this part as a practice for other things we learned so far. Let's dive right in.

First off, we make the HTML file for our not found page somewhat like below, where do we make this file? of course in the master templates directory.

```
<!-- /MyProject/templates/404.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>
<body style="text-align: center; font-size: 100px;">
    <h1 style="line-height: 50px;">404</h1>
    <h3 style="font-size: 60px;">Not Found!</h3>

    <h5>
        {% block url %}
            /
        {% endblock url %}
    </h5>
</body>
</html>
```

Then, it's time to make a view that renders this page.

```
from django.template.loader import render_to_string
from django.http import HttpResponseNotFound

def not_found(request, dynamic):
    response = render_to_string('404.html') # Another way was to
extend the HTML file and user render().
    return HttpResponseNotFound(response)
```

Finally, add a Dynamic Address URL to the `urlpatterns`.

```
from django.urls import path
from MyProject.app1 import views

urlpatterns = [
    path("app1", views.index, name="app1"),
```

```
    path("<dynamic>", views.not_found, name="not_found")
]
```

One cool thing here is that `django.http` has a built in exception called `Http404()` which could be raised in the `not_found()` view we made. This exception looks at mentioned URLs in project settings for a file named exactly `404.html`. and returns that file.

You won't see the result of `Http404()` exception in your project when you're in debug mode!

Literally everything about this exception shouts it's a best practice!

Final Joke!

If you deploy your website in `Debug = True`, start packing your stuff right away and say goodbye tothe company!

Static Files (CSS, JS, mp4, jpg, ...)

To add static files like styles and scripts, we need to follow the same path as when we wanted to make templates. Inside the app directory we make a folder called `static` and inside it we add anotherfolder named the same as our app. The hierarchy for this project is shown below.

```
/MyProject
    /MyProject
    /user
        /templates/user
            index.html
        /static/user
            styles.css
            scripts.js
```

Make sure you have `django.contrib.staticfiles` added to project settings

```
INSTALLED_APPS = [
    'django.contrib.staticfiles',
]
```

The set up for this project is fairly simple. We have a Master Template file with basic HTML element, the `index.html` at user directory extends it and adds a "User Page" h1 title. the `styles.css` holds a background-color for the body element, the `scripts.js` holds a function called `hello()` which logs "Hello, World!" in console. Now let's see how we can render the styles and scripts alongside html!

First off, take a look at project settings, you'll find something like this:

```
STATIC_URL = 'static/'
```

Now let's take a closer look to the Master Template & HTML file we want to render with styles:

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

```
</body>
</html>

<!-- index.html -->
{% extends "master.html" %}
{% load static %} <!-- Pay attention to load -->

{% block head_ref %}
    <link rel="stylesheet" href="{% static 'user/styles.css' %}">
{% endblock head_ref %}


{% block page_title %}
User
{% endblock page_title %}

{% block body %}
<h1>
    User Page
</h1>
{% endblock body %}
```

And that'll do the work.

One not so tiny detail here is that the number of HTML and CSS files increase massively as the project grows. It is very important how they are foldered and named. Basically the entire project hierarchy is one crutial thing to keep in mind. Below you'll find an AI generated documentation of how a standard Django Project Hierarchy must look like.

Best Practices for Naming and Organizing Files

1. **Use Consistent and Descriptive Names**
   - Name files clearly based on their purpose or the feature they support.
   - Avoid overly generic names like `styles.css` or `script.js`.
2. **Keep Files Organized by App**
   - Group files by app to maintain modularity.
   - Place static files (CSS/JS/images) in a `static` directory within each app.
   - Place templates (HTML files) in a `templates` directory within each app.
3. **Namespace Static Files**
   - Use subfolders or prefixes to avoid filename collisions (e.g., `css/base.css` or `js/admin.js`).
4. **Leverage the Django Template and Static Structure**
   - For templates, keep files under the `templates/<app_name>/` directory.
   - For static files, keep them under `static/<app_name>/`.

Example Project Structure

Here's a scalable Django project directory structure:

Let's work through a real-world example of a Django project. Let's imagine we're building a **Learning Management System (LMS)**, with features like managing courses, user profiles, and discussions.

```
lms_project/
├── manage.py
├── lms_project/                        # Project settings directory
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py                     # Project settings
│   ├── urls.py                         # Root URL configuration
│   ├── wsgi.py
├── requirements.txt                    # Dependencies
├── static/                             # Global/static files (shared
assets)
│   ├── css/
│   │   ├── global.css                  # Global styles for the entire
project
│   ├── js/
│   │   ├── global.js                   # Shared JavaScript logic
│   ├── images/
│   │   ├── favicon.ico                 # Shared favicon
├── templates/                          # Shared/global templates
│   ├── master.html                       # Global base template for the
whole project
│   ├── 404.html                        # Custom 404 page
│   ├── navbar.html                     # Shared navbar template
├── courses/                            # App for managing courses
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations/
│   │   ├── __init__.py
│   ├── models.py                       # Models for courses
│   ├── static/                         # Static files for the courses app
│   │   ├── courses/                    # Namespace for app-specific
static files
│   │   │   ├── css/
│   │   │   │   ├── courses.css         # Styles for the courses section
│   │   │   ├── js/
│   │   │   │   ├── courses.js          # JavaScript for course
interactions
│   │   │   ├── images/
│   │   │   │   ├── course_banner.jpg
│   ├── templates/                      # Templates for the courses app
│   │   ├── courses/                    # Namespace for templates
```

```
│   │   │   ├── base.html           # Base template for course pages
│   │   │   ├── course_list.html    # Page listing all courses
│   │   │   ├── course_detail.html  # Individual course page
│   ├── tests.py
│   ├── urls.py                     # URLs specific to courses
│   ├── views.py                    # Views for course functionality
├── users/                          # App for managing user profiles
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations/
│   │   ├── __init__.py
│   ├── models.py                   # Models for user profiles
│   ├── static/                     # Static files for the users app
│   │   ├── users/                  # Namespace for app-specific
static files
│   │   │   ├── css/
│   │   │   │   ├── profile.css     # Styles for user profile pages
│   │   │   ├── js/
│   │   │   │   ├── profile.js      # JavaScript for user profile
actions
│   │   │   ├── images/
│   │   │   │   ├── avatar_default.png
│   ├── templates/                  # Templates for the users app
│   │   ├── users/                  # Namespace for templates
│   │   │   ├── login.html          # User login page
│   │   │   ├── signup.html         # User signup page
│   │   │   ├── profile.html        # User profile page
│   ├── tests.py
│   ├── urls.py                     # URLs specific to users
│   ├── views.py                    # Views for user functionality
├── discussions/                    # App for managing discussions
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations/
│   │   ├── __init__.py
│   ├── models.py                   # Models for discussion threads
and comments
│   ├── static/                     # Static files for the discussions
app
│   │   ├── discussions/            # Namespace for app-specific
static files
│   │   │   ├── css/
│   │   │   │   ├── discussions.css # Styles for discussion pages
│   │   │   ├── js/
│   │   │   │   ├── discussions.js  # JavaScript for discussion
interactions
│   │   │   ├── images/
```

```
│   │   │   │   ├── thread_icon.png
│   ├── templates/                  # Templates for the discussions
app
│   │   ├── discussions/            # Namespace for templates
│   │   │   ├── thread_list.html    # List of discussion threads
│   │   │   ├── thread_detail.html  # Detail page for a single thread
│   ├── tests.py
│   ├── urls.py                     # URLs specific to discussions
│   ├── views.py                    # Views for discussion
functionality
```

Key Features of this Structure

1. **Realistic App Names**
   - Each app corresponds to a real feature of the LMS (e.g., `courses`, `users`, `discussions`).
2. **App-Specific Organization**
   - Each app has its own `static` and `templates` directories, and files are namespaced to avoid conflicts. For example:
     - `static/courses/css/courses.css`
     - `templates/courses/course_list.html`
3. **Global vs. App-Specific Assets**
   - **Global assets** like `global.css`, `global.js`, and `base.html` are stored in top-level `static/` and `templates/` directories.
   - **App-specific assets** are kept within the app's `static/` and `templates/` directories.
4. **URL Modularization**
   - Each app has its own `urls.py` file, which is included in the main `urls.py` using Django's `include()` function. For example:

```python
from django.urls import path, include

urlpatterns = [
    path('courses/', include('courses.urls')),
    path('users/', include('users.urls')),
    path('discussions/', include('discussions.urls')),
]
```

5. **Ease of Scaling**
   - If a new feature, such as notifications, needs to be added, you can simply create a `notifications` app with its own static files and templates.

Thanks to ChatGPT for providing that knowledge, note that a lot of the things you see in here might be too advanced just for now. They'll come clear soon enough.

Master Static Files

Just like the master template, you can have a master style for your project. We did this in the current project by making a folder at the project root `/MyProject/static`. in which we'll store master styles.

In that folder we added a `master.css` file in which we made the page font-family to `Courier New`. Now we must add the mentioned css file to the `master.html` file we made earlier.

```
{% load static %}

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="stylesheet" href="{% static 'master.css' %}">
    {% block head_ref %}{% endblock head_ref %}
    <title>{% block page_title %}Document{% endblock page_title
%}</title>
</head>
<body>
    {% block body %} {% endblock body %}
</body>
</html>
```

One last step is to add the global static files directory to the project settings like this:

```
from MyProject.MyProject.settings import BASE_DIR

STATIC_URL = 'static/'
STATICFILES_DIRS = [
    BASE_DIR / 'static'
]
```