

DTL e Risorse Statiche

Django Template Language e caricamento di risorse statiche

Cosa abbiamo visto fino ad ora

Primi passi in Django

URL routing and resolution con conseguente funzione di risposta

La risposta restituisce un “HttpResponse” object

Abbiamo semplicemente restituito stringhe

Questo causa la creazione di un HTML minimale

Ma i siti che conosciamo...

...non sono così

Seguono regole di formattazione HTML

HTML è un linguaggio di markup che ci permette di inserire contenuti in aree specifiche della pagina web

Tali contenuti saranno ulteriormente personalizzabili tramite tag

HTML basic tags

Tag	Description
<html> ... </html>	Declares the Web page to be written in HTML
<head> ... </head>	Delimits the page's head
<title> ... </title>	Defines the title (not displayed on the page)
<body> ... </body>	Delimits the page's body
<h <i>n</i> > ... </h <i>n</i> >	Delimits a level <i>n</i> heading
 ... 	Set ... in boldface
<i> ... </i>	Set ... in italics
<center> ... </center>	Center ... on the page horizontally
 ... 	Brackets an unordered (bulleted) list
 ... 	Brackets a numbered list
 ... 	Brackets an item in an ordered or numbered list
 	Forces a line break here
<p>	Starts a paragraph
<hr>	Inserts a horizontal rule
	Displays an image here
 ... 	Defines a hyperlink



Run >

Result

```
<!DOCTYPE html>
<html>
<body>

<h1>Questo testo è detto Heading</h1>
<h2>Questo è un heading più piccolo <h2>
<h3>Pensate ad un titolo/sotto-titolo/sotto sotto titolo...</h3>

<p>Questo invece è un paragrafo.</p>

Questo è un testo normale, <br>
e sono in grado di andare a capo

<br><br>

<center>
Io qui invece mi centro un attimo...
</center><br><br>

<i>Ricetta per i file HTML</i> <br>
<ol>
<li> Un cucchiaino di HTML </li>
<li> Un pizzico di CSS </li>
<li> Javascript Q.B. </li>
</ol>

</body>
</html>
```

Questo testo è detto Heading

Questo è un heading più piccolo

Pensate ad un titolo/sotto-titolo/sotto sotto titolo...

Questo invece è un paragrafo.

Questo è un testo normale,
e sono in grado di andare a capo

Io qui invece mi centro un attimo...

Ricetta per i file HTML

1. Un cucchiaino di HTML
2. Un pizzico di CSS
3. Javascript Q.B.

Links utili [HTML]

- <https://www.w3schools.com/html/default.asp>
- <https://developer.mozilla.org/en-US/docs/Web/HTML/Element?retiredLocale=it>

HTML in Django

Se invece di rispondere con una stringa “normale” avessimo risposto con una stringa formattata secondo i canoni di HTML, avremmo avuto un risultato **visivo** migliore

```
def funzione_risposta(request):
```

```
    response = “Ciao, <br> sono andato a capo”
```



```
    return HttpResponse(response)
```

Possibili problemi

Non esattamente comodo

Non facilmente estendibile

Non facilmente riutilizzabile

Un sacco di codice ripetuto (viola la policy **DRY**, don't repeat yourself)

Non esattamente generico quanto vorremmo...

Soluzione: template



I template sono “scheletri” riutilizzabili nella parte di presentazione della nostra web app.

Si basano sul presupposto che le diverse pagine di un sito “si somigliano”

Dal punto di vista implementativo il codice necessario deve essere riutilizzabile e dinamico

Deve dare la possibilità di andare oltre il mero HTML

DTL: Django template language

E' un vero proprio linguaggio.



Basato su “blocchi” anziché “tag”

I file DTL appaiono come file sorgenti statici ma vengono **dinamicamente risolti lato server** e spediti sotto forma di “risultato statico” all’utente client, tramite browser.

Presentano una comoda interfaccia alla parte in Python della nostra webapp

Presentano una comoda interfaccia per accedere ai parametri della nostra richiesta HTTP

Possono essere estesi in vario modo, evitando inutili ripetizioni di boilerplate code

Che tipo di linguaggio?



Un mix tra markup tramite blocchi e logica di programmazione.

In altre parole, ai blocchi che delimitano gli “spazi notevoli” della nostra pagina abbiamo anche dei programming construct che ci permettono l’accesso in lettura e scrittura alle **variabili di contesto**, ma anche di operare istruzioni condizionali e cicliche tipiche dei **linguaggi di programmazione** (e non solo linguaggi di presentazione).

Altre caratteristiche e grammatica di base

[vedere slides anni precedenti]

django 1 - pipenv _ startproject _ settings _ templates _ views _ urls.pdf

django 3 - template engine _ template language _ views.pdf

Altro esempio

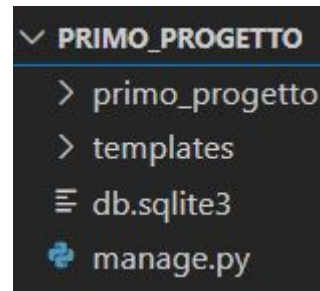
Si parte da un html di base, che sarà utilizzato come template di base.

base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  {% block head %} {% endblock %}
  <title>{% block title %} {% endblock %}</title>
</head>
<body>
{% block content %}
{% endblock %}
</body>
</html>
```

Dove?

Si crei una cartella “templates” nella root del progetto.



In settings.py

```
#attenzione: "os" richiede "import os"
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, "templates")],
        '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',
            ],
        },
    },
]
```

A cosa ci serve?

E' un template base.

Un template in uso ad una risposta può:

- **Estendere** un template base
- **Includere** un template base

Estendiamo il template base

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

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

{% block content %}


    <h1>
        {% if user.is_authenticated %}
            Ciao,  {{ user.username }}
        {% else %}
            Ciao, guest!
        {% endif %}
    </h1>

    {% for i in lista %}

        <p>    {{ i }}    </p>

    {% endfor %}

{% endblock %}
```

questo file è stato chiamato baseext.html 
si trova sempre nella directory root "templates/"

Caratteristiche

```
{% extends "base.html" %}  
  
{% block title %} {{title}} {% endblock %}  
  
{% block content %}  
  
    <h1>  
        {% if user.is_authenticated %}  
            Ciao, {{ user.username }}  
        {% else %}  
            Ciao, guest!  
        {% endif %}  
    </h1>  
  
    {% for i in lista %}  
        <p>    {{ i }}    </p>  
    {% endfor %}  
  
{% endblock %}
```

“extends” statement

ctx variables

session and request variables

programming logic (if/for etc...)

Function view d'esempio

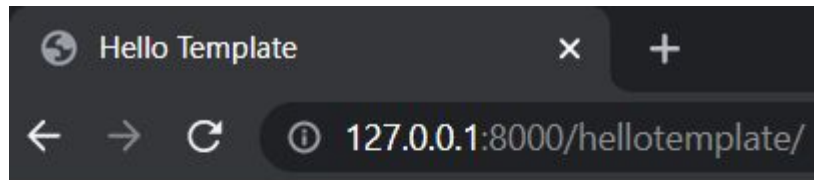
[views.py]

```
def hello_template(request):  
    #occorre importare  
    #from datetime import datetime e  
    #from django.shortcuts import render  
  
    ctx = { "title" : "Hello Template",  
            "lista" : [ datetime.now(), datetime.today().strftime('%A'), datetime.today().strftime('%B')] }  
  
    return render(request, template_name="baseext.html", context=ctx)
```

[urls.py in urlpatterns]

```
path("hellotemplate/", hello_template, name="hellotemplate")
```

127.0.0.1:8000/hellotemplate/



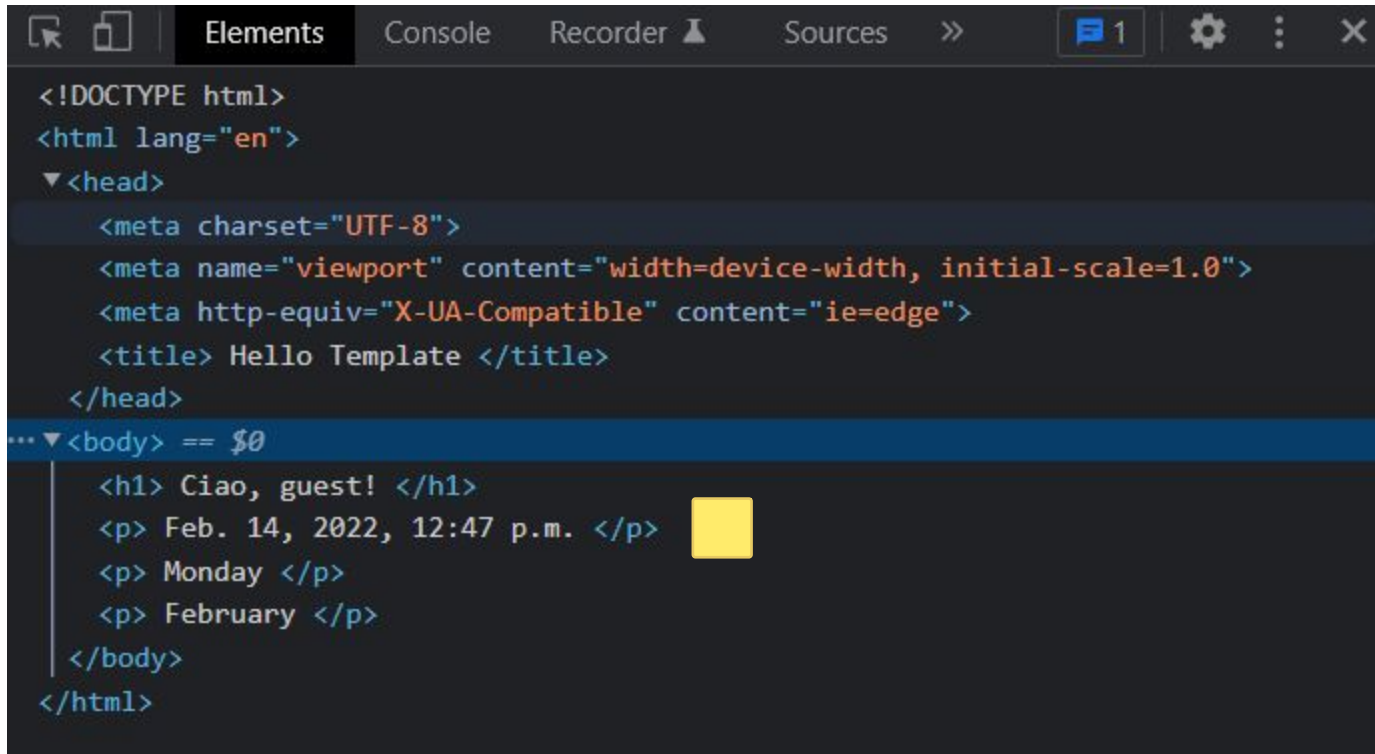
Ciao, guest!

Feb. 14, 2022, 12:45 p.m.

Monday

February

L'HTML generato



The screenshot shows the 'Elements' panel of a web browser's developer tools. The HTML structure is as follows:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title> Hello Template </title>
  </head>
  <body> == $0
    <h1> Ciao, guest! </h1>
    <p> Feb. 14, 2022, 12:47 p.m. </p>
    <p> Monday </p>
    <p> February </p>
  </body>
</html>
```

The `<body>` element is expanded, showing its children. A yellow square is visible next to the second paragraph, `<p> Feb. 14, 2022, 12:47 p.m. </p>`.

DTL: template composition

Oltre a `{% extends "template.html" %}`

Esiste `{% include "template.html" %}` 

Il primo: abbiamo visto come funziona

Il secondo è da intendersi come “copia ed incolla codice **HTML già processato\renderato** all’interno di un altro template.

l’argomento della direttiva include è spesso chiamata “sub-template”.

Esempio

includefooter.html in `<root_prj>/templates/`

```
<footer>
Link utili <br>
<p><a href="http://localhost:8000/welcome_path/Mario/25/">Welcome name, age!</a></p>
</footer>
```

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

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

{% block content %}

    <h1> ...
</h1>

    {% for i in lista %} ...
    {% endfor %}

    {% include "includefooter.html" %}

{% endblock %}
```

cambiamenti nel template file vero e proprio

A proposito link, url path & alias...

Il pezzo di codice precedente (includefooter.html) poteva essere (ri)-scritto così:

```
<footer>  
Link utili <br>  
<p><a href="{% url 'welcomepath' nome='Mario' eta=25 %}">Welcome name, age!</a></p>  
</footer>
```


Attenzione! Cosa significa “include”?

la direttiva “include” deve considerarsi come “rendera questo sub-template e includi il suo HTML generato”.

Il che è diverso da pensare:

“Leggi ed includi il contenuto di questo sub-template ed includilo così come è all'interno del template padre.”

Questo implica che non vi è uno stato condiviso tra template\sub-templates, in quanto vengono renderati in momenti diversi, indipendentemente l'uno dall'altro.

Template inheritance

Estendere blocchi nei template, causa la loro **totale** riscrittura.

In altre parole, se il padre base.html definisce dei blocchi (e.g. block head) con tanto di contenuti, questi ultimi verranno di fatto riscritti nel momento in cui i figli ridefiniscono blocchi con lo stesso nome.

Per capire meglio: base.html

templates > <> base.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  √ <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <meta http-equiv="X-UA-Compatible" content="ie=edge">
7      {% block head %} {% endblock %}
8      <title>{% block title %} {% endblock %}</title>
9  </head>
10 <body>
11 {% block content %}
12 Con il termine lorem ipsum si indica un testo segnaposto utilizzato da grafici,
13 designer, programmatori e tipografi a modo riempitivo per bozzetti e prove grafiche.
14 È un testo privo di senso, composto da parole (o parti di parole) in lingua latina,
15 riprese pseudocasualmente da uno scritto
16 di Cicerone del 45 a.C, a volte alterate con l'inserzione di passaggi ironici.
17 La caratteristica principale è data dal fatto che
18 offre una distribuzione delle lettere uniforme, apparendo come un normale blocco di
19 testo leggibile.
20 {% endblock %}
21 </body>
22 </html>
```

baseext.html

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

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

{% block content %}

    <h1>
        {% if user.is_authenticated %}
            Ciao,  {{ user.username }}
        {% else %}
            Ciao, guest!
        {% endif %}
    </h1>

    {% for i in lista %}

        <p>    {{ i }}    </p>

    {% endfor %}

{% endblock %}
```

← → ↻ ⓘ 127.0.0.1:8000/hellotemplate/

Ciao, guest!

Feb. 14, 2022, 2:17 p.m.

Monday

February

baseext.html

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

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

{% block content %}

    <h1>
        {% if user.is_authenticated %}
            Ciao,  {{ user.username }}
        {% else %}
            Ciao, guest!
        {% endif %}
    </h1>

    {% for i in lista %}

        <p>    {{ i }} </p>

    {% endfor %}

{% endblock %}
```



Ciao, guest!

Feb. 14, 2022, 2:17 p.m.

Monday

February

Il paragrafo su *lorem ipsum* **non** viene “ereditato”

E se volessi ereditarlo?

Esiste l'istruzione:

`{{block.super}}` all'interno di un blocco `{% block nameblock %}...{% endblock %}`



Nell'esempio di prima, inserendo `{{block.super}}` all'interno di `{% block content %}` e `{% endblock %}` dentro `baseext.html`, che già estende `base.html`, saremo in grado di vedere il lorem ipsum che è stato descritto nel template padre.



Leggere i parametri url tramite DTL

Richieste GET,

url_path?nome=value1&eta=value2...

```
<p> {{ request.GET.nome }} </p>
```

```
<p> {{ request.GET.eta }} </p>
```

```
<!-- per controllarli tutti... -->
```

```
{% for key, value in request.GET.items %}
```

```
    {{ key }} = {{ value }}
```

```
{% endfor %}
```

Passaggio parametri tramite URL: lettura in DTL

```
path("helloparams/<str:nome>/<int:eta>/", hello_params_url, name="helloparamurl")
```



```
<p>{{ request.resolver_match.kwargs.nome }} </p>  
<p>{{ request.resolver_match.kwargs.eta }} </p>
```



Risorse Statiche

static in Django

Risorse statiche

Una risorsa statica di una webapp è tutto ciò che non è dinamicamente generato o modificabile dall'utente

Esempi: immagini, video, fogli di stile, JS scripts etc...



Risorse statiche in Django

Malgrado la semplicità del concetto di risorsa statica, in Django gestire le risorse statiche può non essere banale.



E' uno di quegli aspetti in cui l'impostazione **DEBUG=True|False** causa una differenza **significativa nel modo in cui vengono gestiti**.

Perchè?

Non si faccia l'errore di considerare le risorse statiche come i template....

Non sono simili, né dal punto di vista concettuale, né dal punto di vista di come vengono gestiti da Django.

static

In django le configurazioni che contengono la parola *static* hanno a che fare con la gestione delle risorse statiche.

Ecco l'inghippo:

Mentre i template risiedono comunemente nella stessa macchina in cui è presente la logica in Python della nostra web app, **django non fa alcuna assunzione su dove nel mondo siano presenti le vostre risorse statiche.**

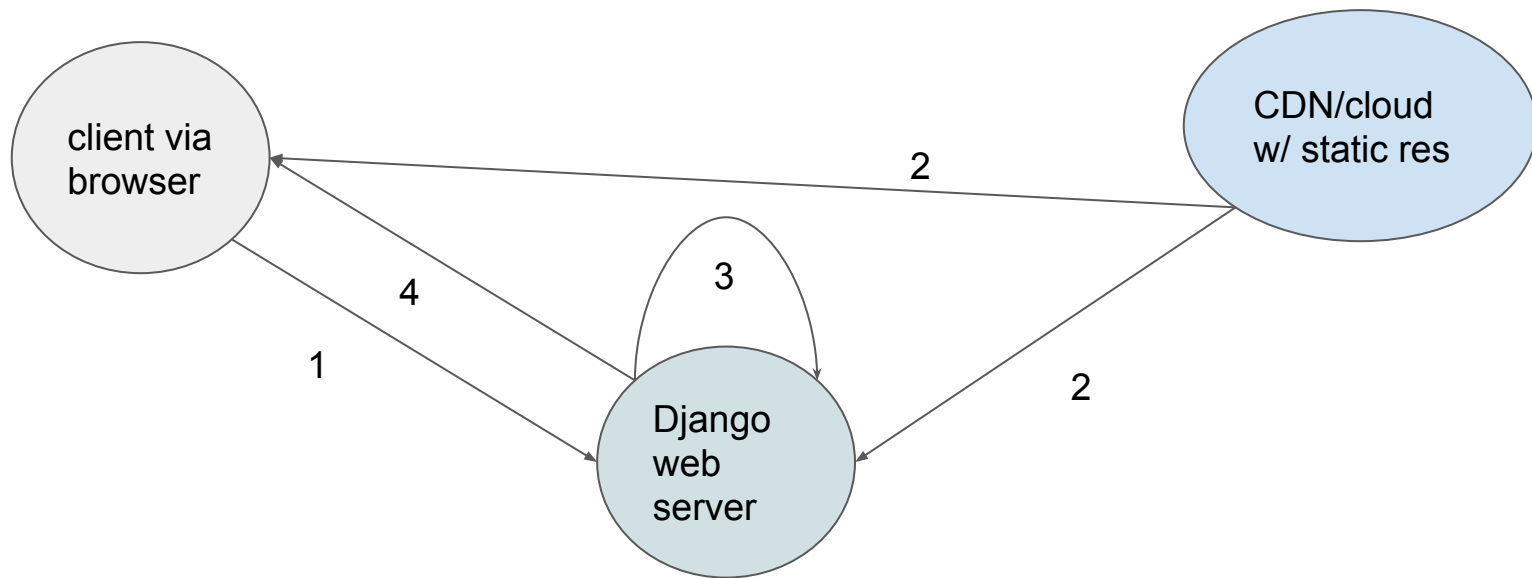
Comunemente

Comunemente, nei production systems,

Le risorse statiche vengono caricate on-demand, e risiedono in

- Servizi cloud
- CDNs, **content delivery networks**
- ... altri host remoti, diversi da dove risiede la logica applicativa...

Possibile soluzione



Descrizione

Un client, tramite richiesta da browser contatta il nostro django web server [1].

La pagina di risposta contiene elementi dinamici, ma anche elementi statici (per es. un'immagine).

Quest'ultima deve essere **servita**, in quanto è collocata in un server apposito. [2]

Quindi, il web server **può** eventualmente avere una cache per questa specifica risorsa\immagine.

Ad ogni modo, l'immagine viene combinata con gli altri elementi dinamici e poi [3] finalmente rispedita al mittente della richiesta iniziale [4].

Servire risorse statiche in production

<https://docs.djangoproject.com/en/4.0/howto/static-files/deployment/>

The easy way: servire file statici se DEBUG=True

in settings.py

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

!



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

Possiamo quindi creare una cartella “static” allo stesso livello di dove abbiamo creato la cartella “templates”. Lì possiamo inserire le nostre risorse statiche



Su DTL template file

Nel blocco “head” di un template in DTL+HTML

```
{% load static %} 
```

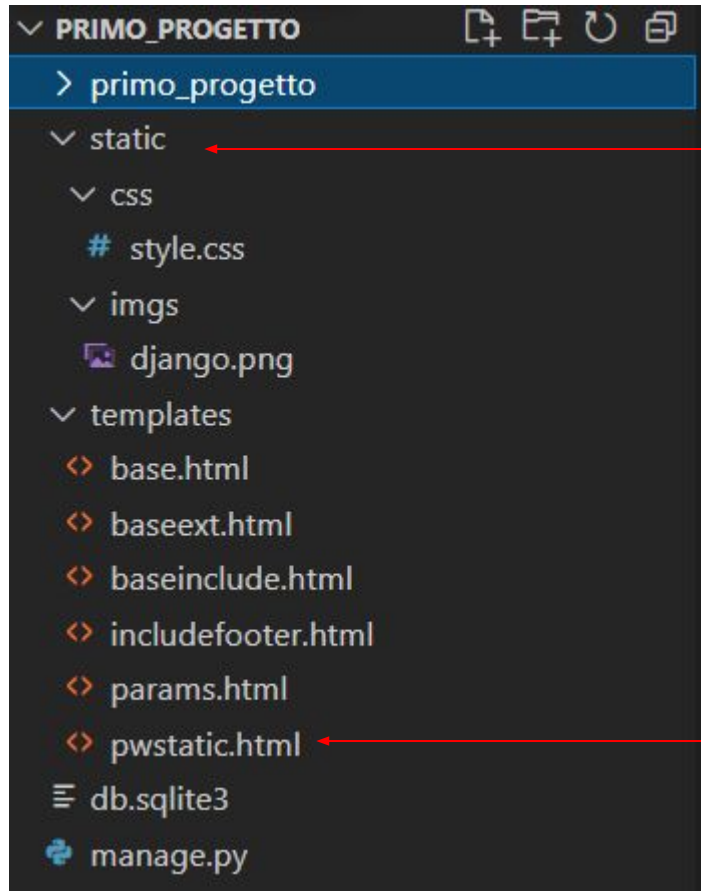
Esempi di utilizzo: 

```

```

```
<link rel="stylesheet" type="text/css" href='{% static "/css/style.css" %}'>
```

Proviamo



static
static/imgs
static/css

DTL template di
prova

pwstatic.html




templates > <> pwstatic.html

```
1  {% extends "base.html" %}
2
3  {% block head %}
4
5  {% load static %}
6
7  <link rel="stylesheet" type="text/css" href='{% static "/css/style.css" %}'>
8
9  {% endblock %}
10
11 {% block title %} {{title}} {% endblock %}
12
13 {% block content %}
14 |
15 <center>
16
17 
18
19 <footer>
20 Link utili su CSS <br>
21 <p><a href="https://www.w3schools.com/css/">CSS Web School </a></p>
22 </footer>
23
24 </center>
25
26 {% endblock %}
```

CSS

Cascading Style Sheets:

<https://www.w3schools.com/css/>

```
static > css > # style.css >  footer
1  footer {
2      text-align: center;
3      padding: 3px;
4      background-color:  salmon;
5      color:  oldlace;
6  }
```

Response view function

```
def page_with_static(request):  
    return render(request,  
                  template_name="pwstatic.html",  
                  context={"title": "Pagina con elementi statici"})
```



Link utili su CSS

[CSS Web School](#)

Alcune considerazioni

Serving static files se DEBUG=True



Serving the files

In addition to these configuration steps, you'll also need to actually serve the static files.

During development, if you use `django.contrib.staticfiles`, this will be done automatically by `runserver` when `DEBUG` is set to `True` (see `django.contrib.staticfiles.views.serve()`).

This method is **grossly inefficient** and probably **insecure**, so it is **unsuitable for production**.

See [How to deploy static files](#) for proper strategies to serve static files in production environments.



Generic vs app template/static files

Sia i template che i file statici:

- Possono essere ad uso e consumo di una app specifica del vostro progetto
- Possono essere generici, e quindi ad uso e consumo di più app o solo del progetto di root

In queste slides ci siamo per ora occupati solo della project root e non abbiamo ancora installato app.

Ma presto useremo le app, e quindi occorrerà introdurre una gerarchia di cartelle template & static a partire **dalle directory delle nostre app**.

Occorrerà inoltre garantire che non vi saranno conflitti a livello di naming...