Programmazione Web

Lez. 15

DJANGO

Giuseppe Psaila

Università di Bergamo

giuseppe.psaila@unibg.it

Installazione

- Verificare che l'interprete Python sia raggiungibile dal propmpt
  comando: **`python`**
- Verificare di avere anche l'utilità **`pip`**, per gestire l'installazione dei package
  Comando: **`pip`**
- Installazione:
  `python -m pip install Django`

Installazione: Verifica

- Avviare l’interprete python da linea di comando
- Eseguire i seguenti comandi:

```
import django
print(django.get_version())
```

- Se compare un messaggio come

```
5.0.6
```

allora DJANGO è installato.

Creare l'Applicazione

- Il comando **`django-admin`** serve per creare le web application.
- Il comando è:
  `django-admin startproject myapp`
- Ma se non esiste, si può ovviare in questo modo
  `python –m django startproject myapp`
- Verrà creata una cartella nella cartella di lavoro corrente del prompt

Avviare il Server

- Abbiamo creato l'app **`myapp`**
- All'interno della cartella omonima, troviamo:
  un'altra cartella **`myapp`**
  il programma **`manage.py`**
- Per avviare
  `python manage.py runserver`

Avviare il Server

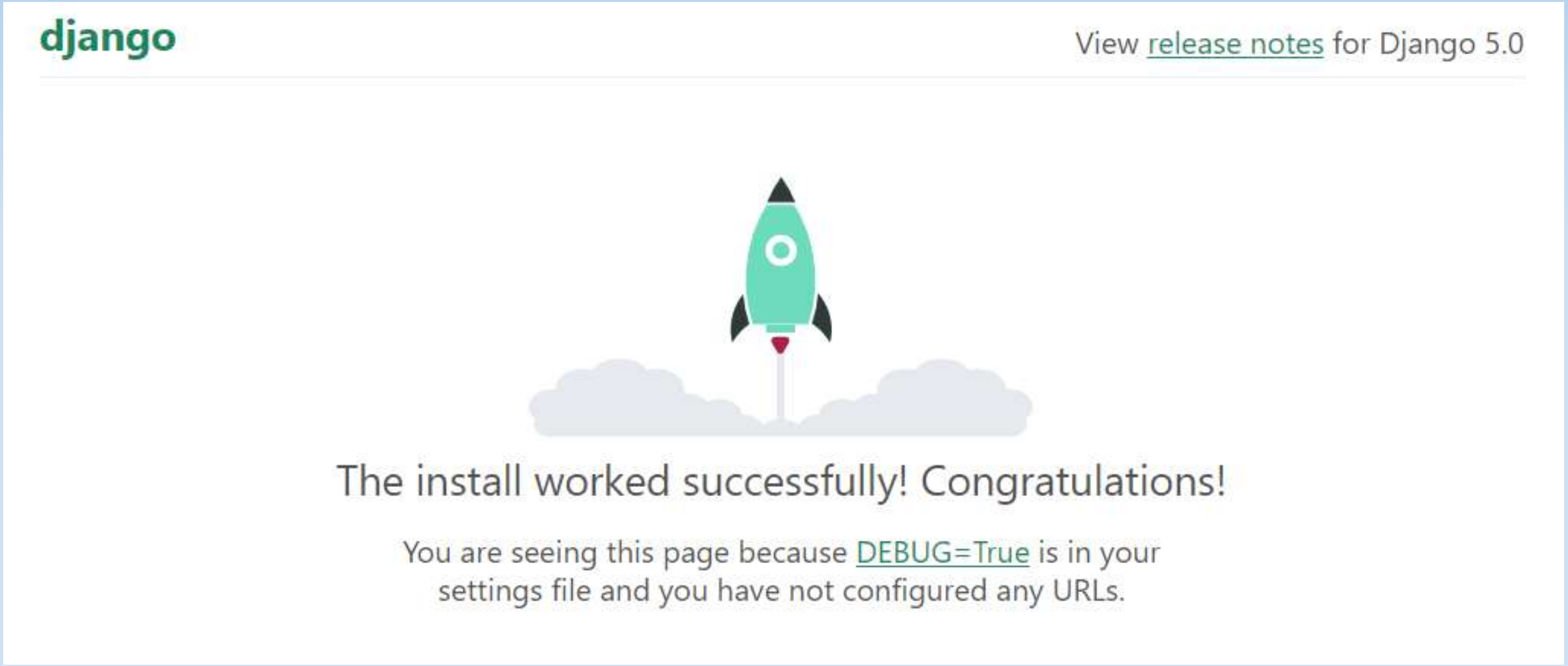
- Dovrebbe comparire il messaggio seguente

```
You have 18 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.
May 17, 2024 - 16:59:16
Django version 5.0.6, using settings
'myapp.settings'
Starting development server at
http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Server Avviato

- Il server è stato avviato sulla porta 8000
- Provando a visualizzare la home page, comapre una pagina di congratulazioni.

Server Avviato



Attenzione

- La documentazione dice chiaramente che il web server in uso non è professionale, ma per supportare lo sviluppo.
- A livello professionale, occorre usare un web server professionale (come **`Apache httpd`**), integrare l'interprete python e DJANGO.

Approccio MVC

DJANGO è organizzato secondo il pattern MVC (Model-View-Controller)

- Il database è il «Model»
- La richiesta HTTP invoca una «funzione Python» (controller)
- La funzione usa un «template», una pagina HTML parametrizzata su un contesto (View)

Come Procediamo

- Prima vediamo un semplice controller
- Poi vediamo come gestire la risposta
- Quindi aggiungiamo le query (PostgreSQL)
- Infine aggiungiamo i template

Controller Semplice

Creiamo il Controller

- Nella cartella **`myapp/myapp`**
- Creiamo il file **`myController.py`**
- Nella slide successiva, vediamo come è fatto

Controller

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    o1 = "<html> <body>"
    o2 = "<p>Welcome to DJANGO</p>"
    o3 = "</body> </html>"
    return HttpResponse(o1 + o2 + o3)
```

Funzione index

- La funzione **`index`** riceve come parametro un oggetto che descrive la richiesta HTTP
- Deve restituire un oggetto che descrive la risposta HTTP, classe **`HttpResponse`**. Dal modulo **`django.http`**
- Il costruttore di **`HttpResponse`** riceve la stringa da inviare.

Mappare Funzione e URL

- Nel file **`urls.py`** (sempre nella cartella **`myapp/myapp`**)
- viene creato il mapping tra un path/url e una funzione in un modulo controller (possono essercene molti).
- Nella prossima slide vediamo come.

Mappare Funzione e URL

```
from django.contrib import admin
from django.urls import path
from django.conf.urls import include

from . import myController

urlpatterns = [
    path("", myController.index,
         name="index")
]
```

Mappare Funzione e URL

- Passo 1: importare il controller
  `from . import myController`

Passo 2: creare/estendere la lista **`urlpatterns`** con oggetti di tipo **`path`**

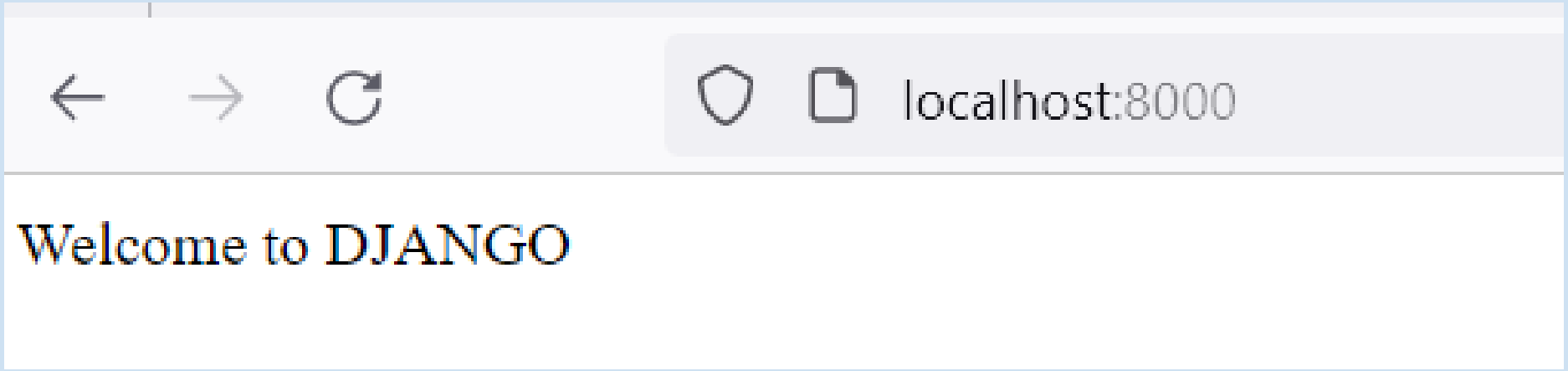
```
urlpatterns = [
    path("", myController.index,
         name="index")
]
```

Mappare Funzione e URL

- L’oggetto path è importato dal modulo
  `django.urls`
  `from django.urls import path`
- Mappa un URL su una funzione in un modulo
  `path("", myController.index,`
  `name="index")`

Provoamo

- Nella barra degli indirizzi del broser
- **`http://localhost:8000/`**



Nomenclatura

- DJANGO segue il pattern MVC a tutti gli effetti.
- Ma quello che noi chiamiamo «**controller**» (che è un controller a tutti gli effetti), nella documentazione DJANGO viene chiamato «**view**».

HttpResponse

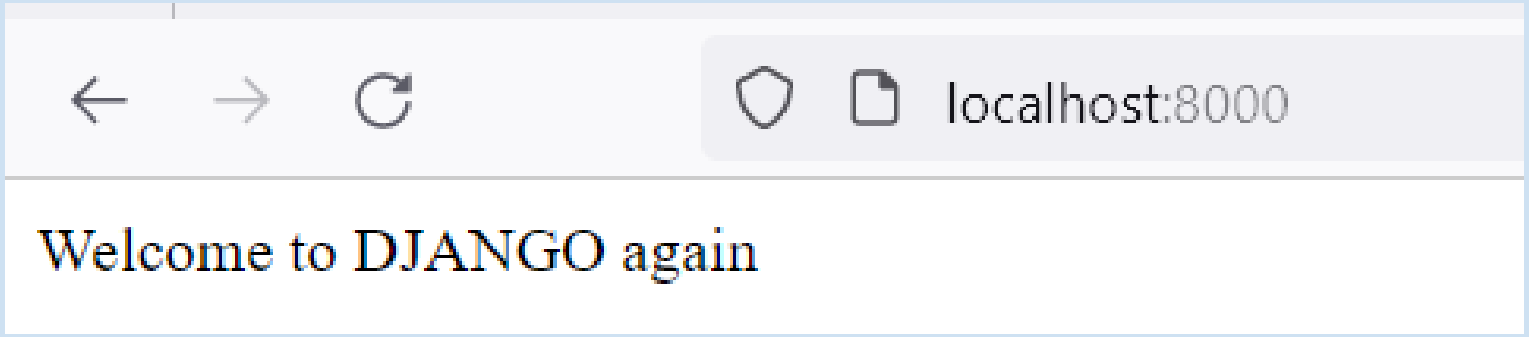
- Il costruttore della classe **`HttpResponse`** può essere invocato senza specificare il contenuto **`HttpResponse`**()
- Perché fornisce il metodo **`write`**, con il quale si può scrivere nella risposta in tempi diversi.
- Nel costruttore, si può specificare un parametro supplementare con il MIME Type della risposta **`HttpResponse(content_type="text/html")`**

HttpResponse

```
def index2(request):
    response = HttpResponse(
       content_type="text/html")
    response.write("<html> <body>")
    response.write(
       "<p>Welcome to DJANGO again</p>")
    response.write("</body> </html>")
    return response
```

Modifichiamo il Mapping

```
urlpatterns = [
    path("", myController.index2,
         name="index")
]
```



HttpRequest

- Il parametro **`request`** della funzione è definito sulla classe **`HttpRequest`**
- Che cosa può servirci della richista:
  Il metodo HTTP
  I parametri della query string (metodo GET)
  Il contenuto della richiesta (metodo POST)
  Il MIME Type della richiesta

HttpRequest: Quale Metodo?

- La proprietà **`request.method`** è una stringa che riporta il nome del metodo HTTP, in maiuscolo

```
if request.method == "GET":
   do_something()
elif request.method == "POST":
     do_something_else()
```

HttpRequest: Metodo GET

- La proprietà **`request.GET`** è un dizionario (o meglio, un oggetto che si comporta come un dizionario) con tutti i parametri della query string
- La classe è **`django.http.QueryDict`**

- Il metodo per ottenere un parametro è **`get(name, default)`**
- Se presente, fornisce il valore del campo **`name`**; **`defailt`** (opzionale) è il valore se il campo non esiste.

HttpRequest: Metodo GET

- Invocando il metodo **`dict`**() di **`django.http.QueryDict`**.
- Si ottiene un dizionario puro.
- Con la scrittura **`request.GET.dict().keys()`**
- Si ottiene la lista dei nomi dei parametri ricevuti nella query string.

HttpRequest: Metodo POST

- La proprietà **`POST`** è ancora un oggetto **`QueryDict`**
- Se il metodo POST viene invocato da una form di HTML, contiene i campi della form.

HttpRequest: MIME Type

- La proprietà **`request.content_type`** è una stringa con il MIME Type del contenuto della richiesta.

HttpRequest: Contenuto

- Con il metodo **`request.readlines()`** si può estrarre il contenuto testuale della rihiesta, ottenendo una lista di righe

Esempio: paramsToJson

- Scriviamo un controller che legge i parametri dalla query string o dal contenuto della richiesta
- e genera in output un documento JSON con i parametri e il loro valore.

Esempio: paramsToJson (1/3)

```
def paramsToJson(request):
    if request.method == "GET":
        params = request.GET
    else:
        params = request.POST
```

Esempio: paramsToJson (2/3)

```
o = {}
for n in params.dict().keys():
    o[n] = params.get(n)
```

Esempio: paramsToJson (3/3)

```
res = HttpResponse(
    content_type="application/json")
res.write(json.dumps(o))
return res
```

Esempio: paramsToJson

- Modifichiamo il file **`urls.py`**

```
urlpatterns = [
    path("", myController.index2,
         name="index"),
    path("paramsToJson",
          myController.paramsToJson,
          name="paramsToJson")
]
```

Esempio: paramsToJson

- Otteniamo
  `http://localhost:8000/paramsToJson?a=Pippo,%20b=Pluto`

```
{"a": "Pippo, b=Pluto"}
```

Sessioni

Preparare l’App

- Nella modalità base, DJANGO gestisce le sessioni trame un DB gestito da SQLLite.
- SQLLite è un DBMS relazionale più snello dei classici DBMS relazionali.
- Per creare il DB per l’applicazione, occorre eseguire il comando:
  `python manage.py migrate`

Oggetto Session

- L'oggetto **`session`** è all'interno della **`request`**.
- Come per le servlet,è automaticamente associato alla **`response`**.
- Per impostare un elemento, si usa la sintassi dei dizionari:
  `request.session["name"]=value`
- Per cancellare l'elemento:
  `del request.session["name"]`

Oggetto Session

- Il metodo get consente di recuperare il valore
- **`request.session.get("name")`** recupera il valore; se non esiste, restituisce **`None`**
- **`request.session.get("name«, defValue)`** se il valore non esiete, resituisce il valore di default **`defValue`**

Esempio

Definiamo tre URL

- **`StartSession`**
  attiva una sessione e inizializza un contatore
- **`CloseSession`**
  distrugge il contattore (chiude la sessione)
- **`SessionCount`**
  incrementa il contatore

Esempio: StartSession

```
def StartSession(request):
    res = HttpResponse(content_type="text/html")

    counter = request.session.get("counter")
    if counter == None:
        request.session["counter"] = 1
        res.write("Session Started")
    else:
        res.write("Session already started")
    return res
```

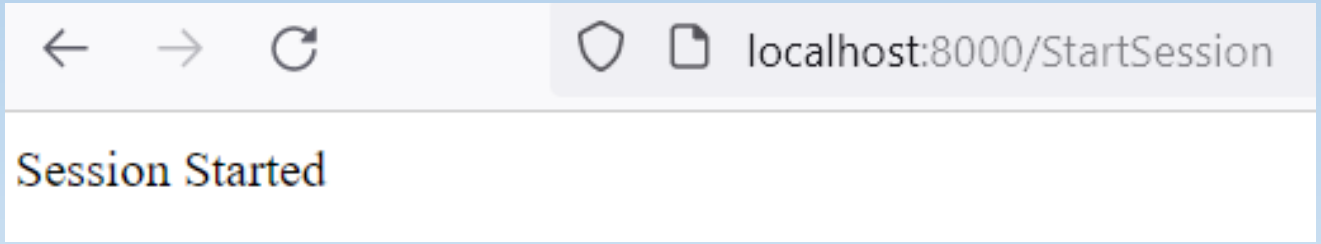
Esempio: CloseSession

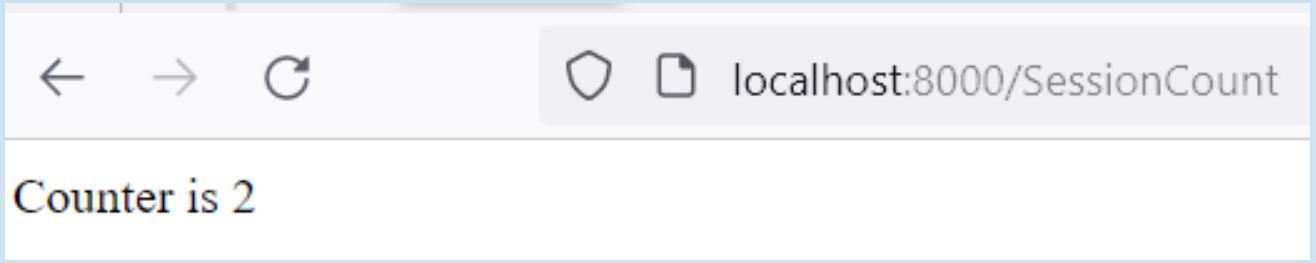
```
def CloseSession(request):
    res = HttpResponse(content_type="text/html")

    counter = request.session.get("counter")
    if counter != None:
        del request.session["counter"]
        res.write("Session stopped")
    else:
        res.write("Session not started")
    return res
```

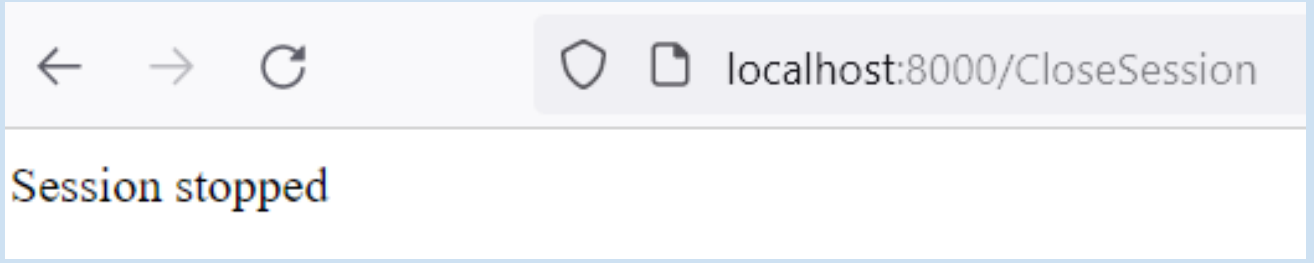
Esempio: SessionCount

```
def SessionCount(request):
    res = HttpResponse(content_type="text/html")

    counter = request.session.get("counter")
    if counter == None:
        res.write("No session activated")
    else:
        counter+=1;
        res.write("Counter is {}".format(counter))
        request.session["counter"]=counter;
    return res
```

Output


localhost:8000/StartSession
Session Started


localhost:8000/SessionCount
Counter is 2


localhost:8000/CloseSession
Session stopped

Templates

Model-View-Controller

- DJANGO adotta il pattern MVC,
- interpretandolo nello stesso modo di ThyeLeaf

- Controller: il codice Python
- View: un file HTML con comandi aggiuntivi (Template)
- Il controller prepara un contesto, che viene usato per elaborare il template

Template Semplice: Hello.html

```
<!DOCTYPE HTML><html lang="en"><head>
<meta charset="UTF-8">
<title>ThymeLeaf Examples</title>
</head>
<body>
<p>Hello {{name}}</p>
</body></html>
```

Caricare il Template

- Per caricare il template ed elaborarlo, occorre importare un «template loader»

```
from django.template import loader
```

- Quindi, il loader carica il tempalte e ne fornisce una rappresentazione interna

```
template =
    loader.get_template("Hello.html")
```

Preparare il Contesto

- Il contesto è un dizionario, che contiene tutte le informazioni da fornire al template.
- Nello specifico, i campi del contesto verranno visti come variabili del tempalte

```
context = { "name": "John" }
```

- Nel template:

```
<p>Hello {{name}}</p>
```

Rendering del Template

- Il metodo **`render`** del template elabora il template e genera l'HTML finale

```
page = template.render(context, request)
```

- L'output va mandato sulla **`response`**

```
res =
 HttpResponse(content_type="text/html")
res.write(page)
return res
```

Output

```
<!DOCTYPE HTML><html lang="en"><head>
<meta charset="UTF-8">
<title>ThymeLeaf Examples</title>
</head>
<body>
<p>Hello John</p>

</body></html>
```

Cartelle dei Template

- Nel file **`settings.py`** vi sono i parametri di configurazione dell'app
- Per indicare in quale cartella andare a cercare i template, cerchiamo l'array **`TEMPLATES`**, che contiene dei dizionari (almeno uno).
- All'interno di questo dizionario, vi è il campo **`DIRS`**, array con la lista di cartelle (nel file system) dove andare a cercare i templates

```
'DIRS':
["C:\\Users\\Utente\\Documents\\Lavoro\\corsi\\Progr_Web\\DJANGO\\myapp\\templates"],
```

Esempio Complesso: Nominativi

Riprendiamo ancora l’esempio dei Nominativi.

- Creiamo un dizionario con all’interno la lista di nominativi
- Quindi usiamo un template che deve iterare e deve avere parti condizionate

Driver di PostgreSQL

- Probabilmente, dovrete re-installare il driver di PostgreSQL
- Perché a suo tempo, lo avevamo installato nell'ambiente di Spyder
- Ora stiamo usando l'interprete standard di Python
- Da linea di comando:
  `python -m pip install psycopg2`

Controller: Connessione

```
def Nominativi(request):
    conn = psycopg2.connect(database="MyDB",
      user='MyUser',   password='MyPwd',
      host='localhost', port='5432')


    conn.autocommit = True
```

Controller: Query

```
cursor = conn.cursor()
query = 'SELECT "Name", "Age" FROM "Names";'

cursor.execute(query)
results = cursor.fetchall()
```

Controller: Struttura Dati

```
l = []
for r in results:
    o = {}
    o["name"] = r[0]
    o["age"] = r[1]
    l.append(o)
data = { "elementi": len(l)}
data["list"]=l
```

Controller: Rendering

```
context = {"res": data}


template =
    loader.get_template("Nominativi.html")
page = template.render(context, request)
```

Controller: Response

```
res = HttpResponse(content_type="text/html")
res.write(page)

return res
```

Prepariamo il Contesto

- Il controller prepara il contesto.
- L'oggetto **`data`** è un dizionario, che replica la struttura dati usata per la versione Java:
  campo **`elementi`**;
  campo **`list`** (lista di dizionari con due campi, **`name`** e **`age`**).
- Il contesto contiene il campo **`res`**.

Il Template

Il template deve:

- creare una tabella con i nominativi, se vi sono dei nominativi;
- scrivere che non ci sono nominativi, se non ve ne sono.

Approccio di DJANGO

- Al contrario di ThymeLeaf, i template di DJANGO sono più tradizionali
- cioè incorporano istruzioni di natura procedurale
- racchiuse nei simboli `{%` e `%}`
- Le istruzioni sono ispirate a quelle di Python, ma non potevano basarsi sull'indentazione
- Conseguenza: i blocchi devono essere chiusi

Cuore del Template

```
{% if res.element == 0 %}
<div>
<p>Nessun nominativo trovato</p>
</div>
{% else %}
<table>
…
</table>
{% endif %}
```

Istruzione Condizionale

- È la solita istruzione if di Python
- Non ci sono più i due punti alla fine della riga, perché la prima parte dell'istruzione è delimitata
  `{% if res.element == 0 %}`
- Mancando l'indentazione, deve essere terminata da
  `{% endif %}`

Istruzione Condizionale

- Nel mezzo, troviamo
  `{% else %}`
- Ma potremmo trovare
  `{% elif` *cond* `%}`

Cuore del Template

```
<table>
{% for item in res.list %}
<tr>
<td>{{item.name}}</td>
<td>{{item.age}}</td>
</tr>
{% endfor %}
</table>
```

Iterazioni

- Ritroviamo l’istruzione for di Python
  `{% for item in res.list %}`
- con la chiusura
  `{% endfor %}`

Cuore del Template

```
<table>
{% for item in res.list %}
<tr>
<td>{{item.name}}</td>
<td>{{item.age}}</td>
</tr>
{% endfor %}
</table>
```

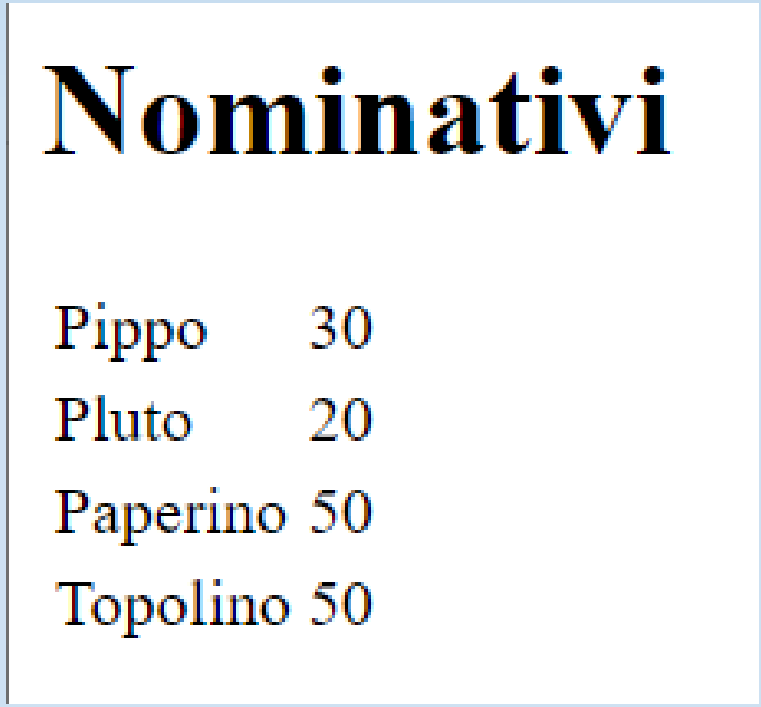
Output dei Valori

`{{item.name}}`

- La doppia graffa dice che il vlore dell’espressione deve essere mandato in output.

Output

- L’output che otteniamo è lo stesso della versione ThymeLeaf



File Accessori (statici)

- Dove mettere file accessori che contengono css, JavaScript, ecc.?
- Secondo DJANGO, sono file «statici», quindi vanno messi in apposite cartelle sotto un URL specifico.
- Vediamo come impostare il file «**`settings.py`**».

File Accessori (statici)

- **`STATIC_URL = 'static/'`**
  Imposta l'URL che contiene tutti i file statici. Per invocarli, si dovrà usare «**`/static/stile.css`**»
- **`STATICFILES_DIRS = [BASE_DIR / 'mystaticfiles']`**
  Descrive tutte le cartelle dove DJANGO va a cercare un file statico.

Considerazioni Finali

Impressioni sui Template

- Finché usiamo i template in questo modo,
- le cose sono semplici e risulta facile gestire i template.
- Ma il linguaggio è molto ricco e consente di aggregare frammenti di template, cosa che rende molto poco leggibile il template stesso.

Impressione Generale

- Nel complesso, DJANGO è una soluzione molto interessante
- perché è basata in modo nativo sul pattern MVC.
- Inoltre, perché ancora una volta Python dimostra di essere un linguaggio di programmazione molto efficiente (velocità di scrittura del codice).

Tutto Bello?

- Ogni soluzione tecnica ha i suoi pro e i suoi contro.
- Io ho trovato complicato gestire contenuti che non devono passare dal controller, per configurare l’app per rendere visibile all’esterno file html, css, JavaScript da includere nelle pagine.
- Inoltre il linguaggio dei Template è molto ricco e occorre molto tempo per padroneggiarlo tutto.

File Django.zip

- Sul team trovate il file **`Django.zip`**, nel quale ho messo tutti gli esempi visti in questa lezione.