

# PYTHON ADVANCED

Curs interactiv de python

# STRUCTURA SEDINTA 8 : DJANGO

▤ Intro Django

▤ Proiect Django

▤ Meniu

▤ Adaugarea de CSS si JS la proiect

# DJANGO: INTRO

Django :

- un Open Source web framework care permite construirea de aplicații web
- creat de Adrian Holovaty si Simon Willison în 2003.
- utilizat pentru Instagram, Mozilla, The Washinton Times, BitBucket, Disqus, Pinterest, Nasa, afnity.com, chess.com, theonion.com
- nu este un limbaj de programare sau nu are un limbaj separat cum se regaseste în kivy

# DJANGO: INTRO



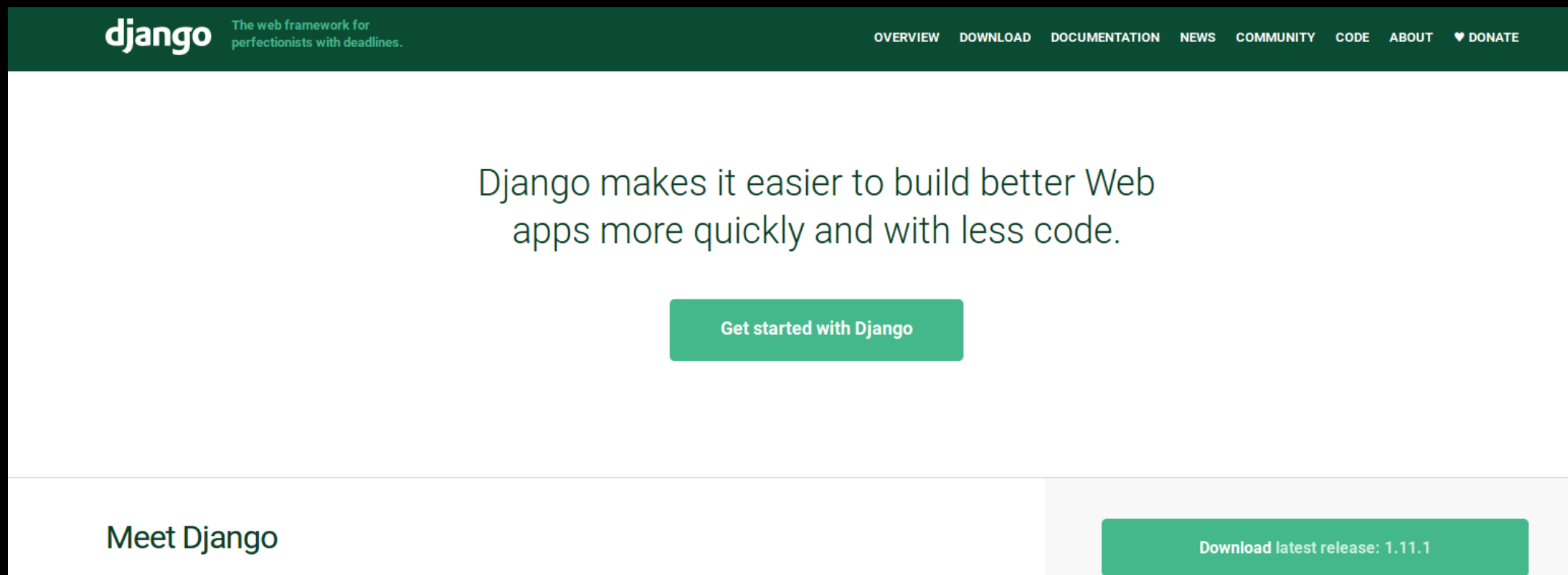
- Django este ORM – object relational mapping
- Django folosește URL Routing
- Django are un sistem de HTML templating
- Django permite un form handling
- DRY – Don't Repeat Yourself - se încearcă eliminarea repetărilor, deci o implementare o singura data.
- Django încearcă o dezvoltare cât mai rapidă
- Suport de Framework – Django are suport inclus de Ajax, caching și alte framework-uri variate.
- Django este integrat in Eclipse, PyCharm, Visual Studio etc.
- Django are o interfața integrată pentru activități administrative.

# DJANGO: INTRO



- De reținut ca Django nu este un web server. Are unul inclus, dar în producție se folosește de obicei Apache sau IIS.

Site-ul oficial al dezvoltatorului este <http://www.djangoproject.com>

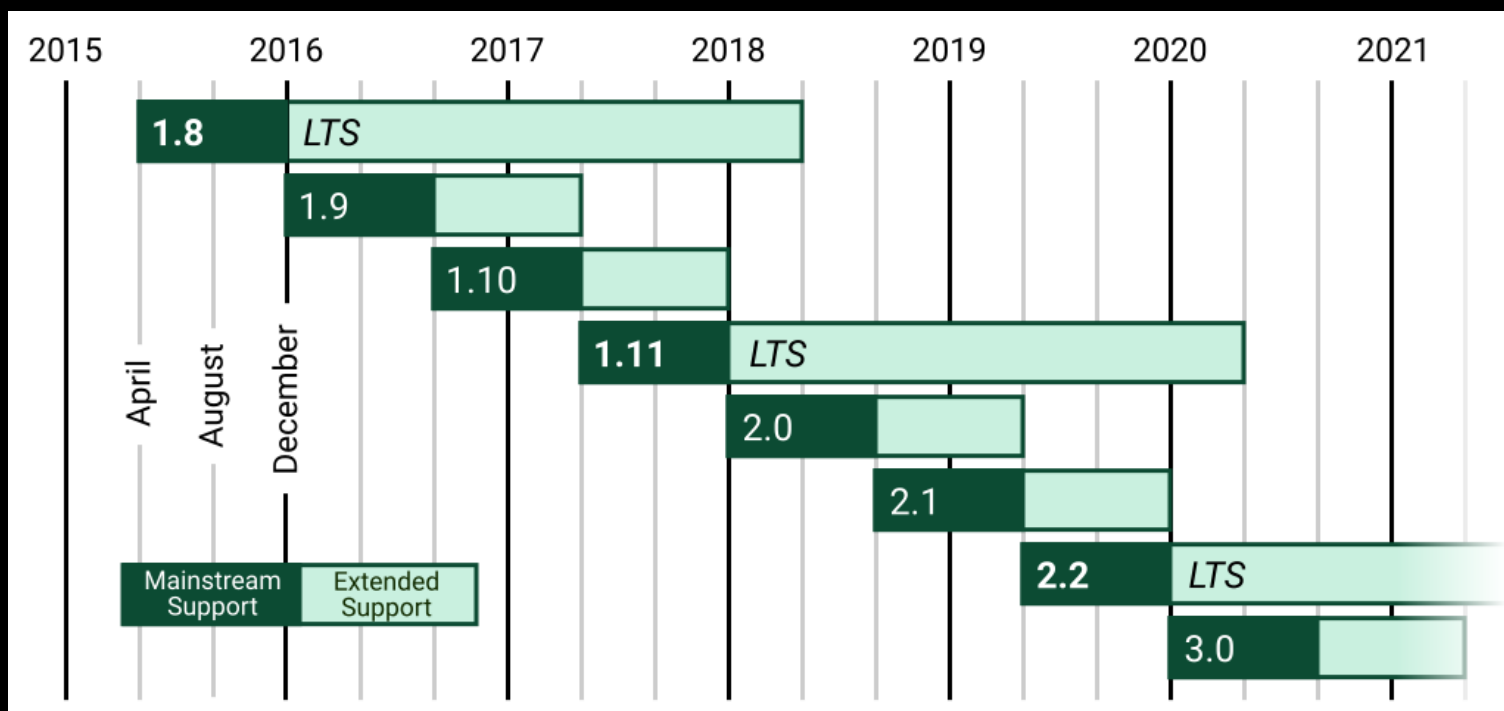


# DJANGO: INSTALARE

- Django este un modul non-standard deci necesita o instalare prin pip.

pip install django==1.11

Dacă apălați comanda pip install django==1.11 indicați ce variaanta trebuie instalata, In momentul scrierii este disponibila variaanta 1.11.1, dar care nu e stabila.



# DJANGO: INSTALARE



- Django este un modul non-standard deci necesita o instalare prin pip.

```
pip install django==1.11
```

Dacă apelați comanda `pip install django==1.11` indicați ce variantă trebuie instalată,

În momentul scrierii este disponibilă varianta 1.11.1, dar care nu e stabilă fiind un release intermediar.

```
C:\Python27\Scripts>django-admin.exe --version  
1.11
```

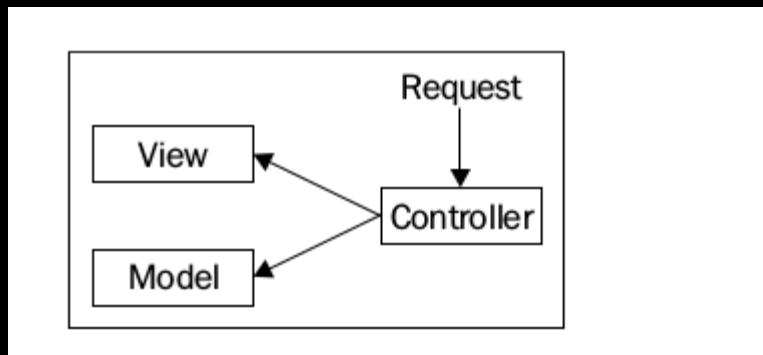
```
C:\Python27\Scripts>
```



# MCV PATTERN ÎN DEZVOLTAREA WEB

MCV Pattern se traduce în Model-View -Controller Pattern:

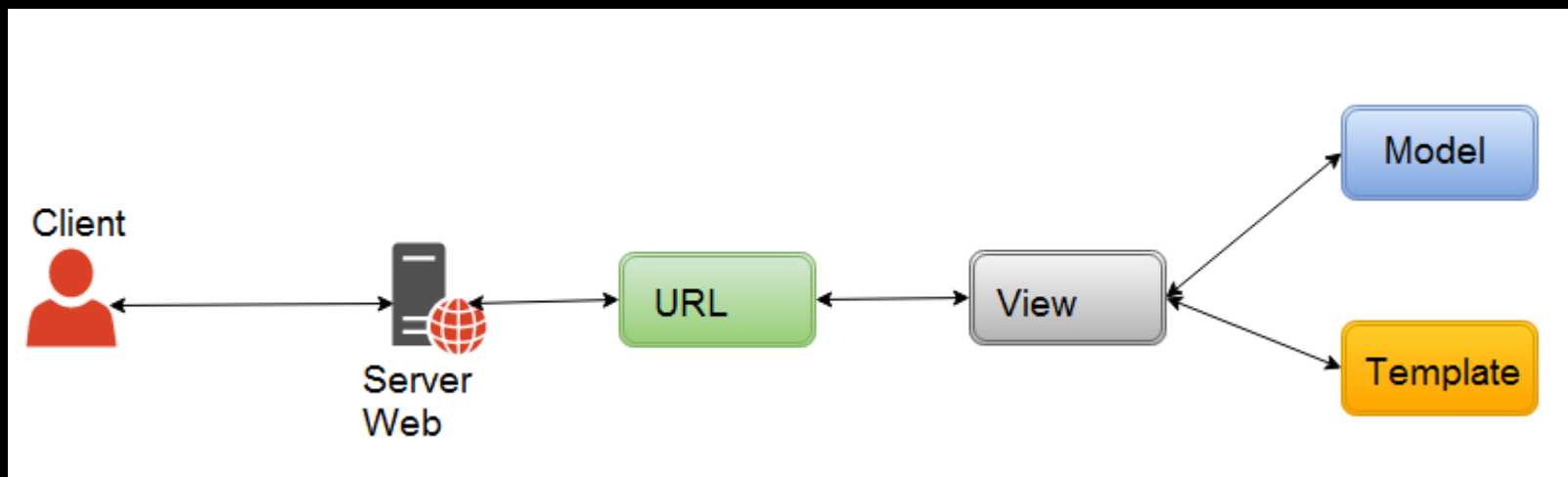
- Model - reprezintă un obiect pentru interfatarea cu datele. Prin urmare Model va interfata cu o baza de date.
- View - reprezintă o modalitate de a vizualiza datele pe care Model le conține.
- Controller – controleaza ambele Model și View. Controlează curgerea de date și din obiectul model și modifica views de fiecare data când este necesar.





# MCV PATTERN ÎN DEZVOLTAREA WEB

Django MCV Pattern este un model derivat de MCV Pattern; Django integreaza rolul controller-ului. URL a aparut in vederea atingerii scalabilitatii deoarece se permit mai multe structuri arborescente de tip VIEW-MODEL-TEMPLATE prin URL. In schimb se introduce conceptul de template care este o extensie vizuala a lui view folosind o combinatie intre HTML si DTL (Django Template Language)



# PROIECT NOU



Pentru a crea un proiect nou trebuie apelat executabilul django-admin.exe ce se regăsește în c:\Python27\Scripts.

Se va apela **django-admin.exe startproject catalog**

Comanda are cuvântul catalog la final indicând numele proiectului nostru.



A screenshot of a Windows Command Prompt window. The title bar is orange and says "Command Prompt". The window shows the current directory as "D:\Catalin\Predare Python\Python curs nou - bitacad\django\_proj". The user has entered the command "C:\Python27\Scripts\django-admin.exe startproject catalog".

```
C:\Python27\Scripts\django-admin.exe startproject catalog
```

# PROIECT NOU



În urma acestei comenzi în directorul unde noi am aplicat aceasta comanda se poate observa apariția subdirectorului catalog.

This PC > All (D:) > Catalin > Predare Python > Python curs nou - bitacad > django_proj > catalog >				
Name	Date modified	Type	Size	
 catalog	5/30/2017 10:45 PM	File folder		
 manage.py	5/30/2017 10:45 PM	Python File	1 KB	

# STRUCTURA PROIECTULUI



Astfel structura devine cea de jos:

catalog/

manage.py

catalog/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

Sa discutam un pic despre fiecare fisier în parte.

manage.py – Mai devreme spuneam ca django se ocupa de controller. Face acest lucru prin manage.py. Deci manage.py devine interfațarea cu django controllers care oferă posibilitatea de a aplica comenzi în vederea rezolvării de sarcini administrative cum ar fi sincronizarea cu o baza de date sau pornirea aplicației.

# STRUCTURA PROIECTULUI







Subdirectorul catalog devine package-ul python ce conține tot proiectul nostru.

Acesta conține un fișier `__init__.py` fără informație

Fișierul `settings.py` conține setările proiectului, așa cum spune și numele.

Fișierul `urls.py` conține diferite link-uri ale proiectului. Django folosește route linking așa cum am menționat mai devreme.

Fișierul `wsgi.py` – setari ale wsgi-ului. Web Server Gateway Interface (WSGI) reprezintă un standard pentru interfete simple și universale între servere web și frameworkuri sau aplicații web scrise în Python – ramane nesetat

This PC ▸ All (D:) ▸ Catalin ▸ Predare Python ▸ Python curs nou - bitacad ▸ django_proj ▸ catalog ▸ catalog			
Name	Date modified	Type	Size
 <code>__init__.py</code>	5/30/2017 10:45 PM	Python File	0 KB
 <code>settings.py</code>	5/30/2017 10:45 PM	Python File	4 KB
 <code>urls.py</code>	5/30/2017 10:45 PM	Python File	1 KB
 <code>wsgi.py</code>	5/30/2017 10:45 PM	Python File	1 KB

# RULAREA PROIECTULUI



Pentru a porni serverul trebuie sa rulam manage.py și sa dam argumentul runserver:

**manage.py runserver**

```
\catalog>manage.py runserver
Performing system checks...

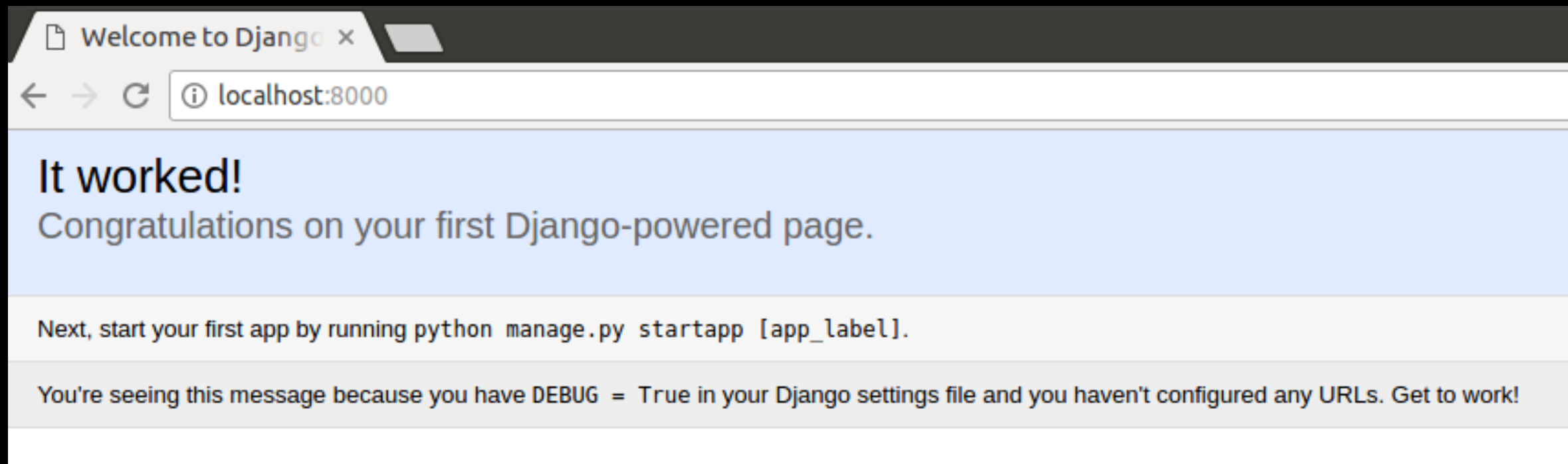
System check identified no issues (0 silenced).

You have 13 unapplied migration(s). Your project may not work properly until you apply the mi
grations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
May 31, 2017 - 10:32:53
Django version 1.11, using settings 'catalog.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

# RULAREA PROIECTULUI



Fișierul `manage.py` ruleaza un server web pentru noi în vederea testării. Astfel ca, la apelarea URL-ului `http://localhost:8000/` putem vedea o pagina care arata ca cea de mai jos.





# RULAREA PROIECTULUI



Dacă citim textul din interior putem vedea ca ni se solicita rularea comenzii **manage.py startapp cuvânt** astfel ca următoarea discuție explica ce este un app.

Un alt aspect important este ca la prima rulare django creează o baza de date sqlite în directorul unde manage.py exista. Prin urmare exista necesitatea unei discuții și pe acest segment.

s\Materiale_studiu\Python_Database\Python_curs_nou_bitacad\django_proj\catalog				
Name	Size	Type	Date Modified	
catalog		File Folder	5/31/2017 10:24 AM	
db.sqlite3	3 KB	SQLITE3 File	5/31/2017 10:24 AM	
manage.py	1 KB	Python File	5/30/2017 10:45 PM	

# RULAREA PROIECTULUI



De asemenea este necesara o observație în a configura rularea serverului pe un alt port decât cel default adică 8000.

Prin aplicarea comenzii **manage.py runserver 0.0.0.0:1234** se va muta portul de acces al serverului pe 1234.

Astfel URL-ul de acces devine `http://localhost:1234/`

În cazul în care apelam **manage.py runserver 0:1234** va face același lucru, 0 reprezentând o scurtatura a lui 0.0.0.0

O a doua observație reprezintă schimbarea fișierelor. Serverul web nu necesita restartarea dacă se vor schimba fișierele proiectului.

# SETAREA UNUI APP



Django este o suma de aplicații. Fiecare aplicație are un obiectiv și poate fi reutilizată în alt proiect, deci se poate privi ca fiind similar cu un package în python.

Pentru a crea o noua aplicație rulați comanda de mai jos:

```
manage.py startapp main
```

Numele aplicației va fi aici **main**, dar ar putea fi setat la orice va doriți.

# SETAREA UNUI APP

După rularea comenzii se poate vedea apariția unui nou subdirector în directorul catalog.

\\Materiale\_studiu\\Python\_Database\\Python\_curs\_nou\_bitacad\\django\_proj\\catalog

Name ▲	Size	Type	Date Modified
catalog		File Folder	5/31/2017 10:24 AM
main		File Folder	5/31/2017 12:07 PM
db.sqlite3	3 KB	SQLITE3 File	5/31/2017 10:24 AM
manage.py	1 KB	Python File	5/30/2017 10:45 PM

\\Materiale\_studiu\\Python\_Database\\Python\_curs\_nou\_bitacad\\django\_proj\\catalog\\main

Name ▲	Size	Type	Date
migrations		File Folder	5/31
__init__.py	0 KB	Python File	5/31
admin.py	1 KB	Python File	5/31
apps.py	1 KB	Python File	5/31
models.py	1 KB	Python File	5/31
tests.py	1 KB	Python File	5/31
views.py	1 KB	Python File	5/31

# SETAREA UNUI APP



Dacă navigam în directorul main acolo regăsim un subdirector numit migrations și o serie de fișiere. Sa discutăm despre fiecare fișier în parte.

`__init__.py` - utilizat pentru a crea un package

`admin.py` – interfata administrativa care permite modificari

`models.py` – aici sunt stocate modelele aplicației, sfors vom gsvr query și ce tabele avem

`tests.py` - teste pe care le poți include aici (optional). Aceste vor fi teste automate.

`views.py` – aici se vor regăsi views-urile care comunica cu modele și template-uri. Prin urmare un strat de control (Control Layer)

Directorul migrations conține fișiere create automat cu schimbările la baza de date într-o forma istorica. Ne vom întoarce aici la momentul potrivit.

# SETAREA UNUI APP

Pentru ca proiectul sa recunoască o aplicație noua trebuie modificat fișierul settings.py.

Astfel ca la linia 33 din fișierul settings.py gasim o lista de siruri de caractere numita INSTALLED\_APPS.

Aici trebuie adăugat numele aplicației pe care dorești sa o adaugi. Astfel ca noi trebuie sa inseram la final o intrare cu sirul de caractere „main”

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

Ln: 33 Col: 18

# SETAREA UNEI BAZE DE DATE



Tot în fișierul settings.py se găsesc și alte setari cum ar fi setarea unei baze de date. Django permite integrarea cu multiple tipuri de baze de date. Oficial exista suport pentru 4 tipuri:

- SQLite,
- MySQL,
- PostgreSQL,
- Oracle.

Totuși se poate folosi și Microsoft SQL Server ce poate fi folosit prin package-ul django-mssql, În timp ce pentru IBM DB2, SQL Anywhere și Firebird exista plugin-uri externe.

Pentru MongoDB și Google App Cloud DataStore se poate folosi NoSQL.



# SETAREA UNEI BAZE DE DATE



În cazul în care avem un SQLite atunci la linia 76-77 găsim un dicționar numit DATABASES care păstrează formatul standard. În cadrul site-ului curent vom utiliza sqlite.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
    }  
}
```

# SETAREA UNEI BAZE DE DATE

Sirul de caractere 'db.sqlite3' se poate schimba în cazul în care avem postgresql atunci acel dicționar imbricat trebuie sa arate ca în figura de mai jos. De asemenea, se poate înlocui șirul de caractere: 'django.db.backends.postgres' cu unul din cele două sirui de caractere de mai jos după necesitatea proiectului :

'django.db.backends.mysql'

'django.db.backends.oracle'

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'mydatabase',  
        'USER': 'mydatabaseuser',  
        'PASSWORD': 'mypassword',  
        'HOST': '127.0.0.1',  
        'PORT': '5432',  
    }  
}
```

Un Model reprezintă posibilitatea unei aplicații să interacționeze cu baza de date. Astfel structura unui fișier `models.py` definește structura bazei de date și definește query-urile.

Structura este foarte flexibilă, fiecare app are propriul `models.py`. Fiecare `models.py` are mai multe clase. Fiecare clasă reprezintă un tabel. Clasa are variabile ale clasei. Fiecare variabilă a clasei reprezintă o coloană în tabel.

Clasele din `models.py` trebuie să moștenească clasa `Model` din `django.db.models.Model`

# MODELS.PY

În proiectul nostru adaugăm în momentul de fata clasa Student ca în figura de mai jos.

```
from __future__ import unicode_literals

from django.db import models

# Create your models here.

class Student(models.Model) :
    nume=models.CharField(max_length=30)
    prenume=models.CharField(max_length=30)
    description=models.TextField()
    proiect=models.FileField(upload_to='proiecte/')
    email=models.EmailField(unique=True)
    nota=models.IntegerField(default=0)
```

# MODELS.PY



Clasa Student moștenește `django.db.models.Model` și are 6 variabile ale clasei.

- Variabila `nume` care indica un `models.CharField()`. `CharField` indica stocarea unui sir de caractere de o dimensiune limitata per intrare, aici 30 de caractere. Limitarea se face prin `max_length` parametru obligatoriu pentru un `CharField`.
- Variabila `prenume` care indica un `models.CharField()`.
- Variabila `description` care indica un `models.TextField()`. `TextField` indica stocarea unui sir de caractere nedefinit de mare.
- Variabila `proiect` care indica un `models.FileField()`. `FileField` indica stocarea unui fișier trimis de utilizator prin metoda `post`. Un alt lucru pe către trebuie să-l știți este ca numele fișierului este stocat în baza de date și poate fi de 100 de caractere lungime. Fișierul va fi stocat în directorul proiectului în mod standard.

Aceste doua caracteristici (lungimea fișierului și directorul care stochează fișierele) se pot schimba. Pentru a schimba calea unde este salvat se folosește atributul `upload_to`

- Variabila email care indica un `models.EmailField()`. `EmailField` are setat default `max_length=254` și validează dacă șirul de caractere pastrează formatul unui email sau nu. Câmpul `unique=True` va indica ca trebuie sa verifice unicitatea acestui câmp înainte ca o noua intrare să se insereze.
- Variabila nota care indica un `models.IntegerField()`. `IntegerField` este un numar din range-ul -2147483648 to 2147483647.

## **Exista diverse attribute ce pot fi folosite în conjunctie cu Field-uri:**

- `default = valoare`      Setăm valoarea default a intrării. Dacă nu e specificata atunci va fi setata aceasta valoare
- `unique=True`      Standard aceasta valoare este False. Se permite adăugarea unei valori exclusiv dacă aceasta nu mai exista într-o alta intrarea
- `index=True`      Default este False. Permite setarea de indecși conform cu conceptul din baze de date.



Când vine vorba de aplicat schimbările ce trebuiesc aplicate bazei de date acestea se fac în trei pași. Primul pas este popularea fișierelor models.py. Acest pas deja a fost realizat.

Urmează crearea unor fișiere ce conțin schimbările ce trebuiesc implementate bazei de date. Trebuie aplicată comanda **manage.py makemigrations**

**Aceste fișiere se regăsesc în main\migrations, adică aplicație\migrations.**

```
catalog>manage.py makemigrations
Migrations for 'main':
  main\migrations\0001_initial.py
  - Create model Student
```

```
D:\Catalin\Predare Python\Python curs nou\
catalog>
```



Ultimul pas pentru aplicarea unor schimbări bazei de date reprezintă rularea fișierelor create în pasul anterior într-un mod automat prin rularea comenzii

## **manage.py migrate**

```
catalog>manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, main, session
s
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying main.0001_initial... OK
  Applying sessions.0001_initial... OK
```

# MODELS.PY

Mai jos putem vedea cum arata baza de date sqlite după schimbările aplicate automat de djangoo.

Database Structure			Browse Data	Edit Pragmas	Execute SQL
Create Table			Modify Table		
Delete Table					
Name	Type	Schema			
auth_permission		CREATE TABLE "auth_permission" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "			
auth_user		CREATE TABLE "auth_user" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "passwo			
auth_user_groups		CREATE TABLE "auth_user_groups" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,			
auth_user_user_permissions		CREATE TABLE "auth_user_user_permissions" ("id" integer NOT NULL PRIMARY KEY AUTOINC			
django_admin_log		CREATE TABLE "django_admin_log" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT			
django_content_type		CREATE TABLE "django_content_type" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMEI			
django_migrations		CREATE TABLE "django_migrations" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT			
django_session		CREATE TABLE "django_session" ("session_key" varchar(40) NOT NULL PRIMARY KEY, "sessio			
main_student		CREATE TABLE "main_student" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "nu			
id	integer	`id` integer NOT NULL PRIMARY KEY AUTOINCREMENT			
nume	varchar(30)	`nume` varchar(30) NOT NULL			
prenume	varchar(30)	`prenume` varchar(30) NOT NULL			
description	text	`description` text NOT NULL			
email	varchar(254)	`email` varchar(254) NOT NULL UNIQUE			
nota	integer	`nota` integer NOT NULL			
proiect	varchar(100)	`proiect` varchar(100) NOT NULL			
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)			

# ALTE FIELD-URI



BooleanField -True/False- Asociat unui CheckBox

FloatField – un float din Python

ImageField – reprezintă un FileField care valideaza dacă fișierul este o imagine.

GenericIPAddressField - Un text introdus de la tastatura ce va fi validat ca o adresa Ipv4 sau IPv6.

SmallIntegerField - un integer intre -32768 și 32767

PositiveSmallIntegerField – un integer intre 0 și 32767

URLField – un CharField care valideaza un URL.

Exista diverse attribute ale câmpurilor pe care nu le pot acoperi în întregime sau opțiunile lor. Exista o relaționare între acestea, unele attribute sunt unice la un field sau nu se pot aplica attribute precum default. Astfel dacă v-ați dori sa le citiți atunci acestea se regăsesc **aici**

Django oferă o interfață gata de utilizare pentru activități administrative, deci și o posibilitate de a interacționa cu baza de date într-un mod facil.

Pentru a putea folosi aceasta interfață este necesar ca noi să cream un user cu drepturi depline. În acest sens vă rog să aplicați comanda

**manage.py createsuperuser**

După aplicarea acestei comenzi veți fi întrebat ce user și parola doriți pentru contul de superuser. Parola trebuie să fie de 8 caractere combinate.

```
\catalog>manage.py createsuperuser
Username (leave blank to use 'administrator'): admin
Email address: catalin.popescu@bitacademy.net
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Password:
Password (again):
Error: Your passwords didn't match.
Password:
Password (again):
Superuser created successfully.
```

Django oferă o interfață gata de utilizare pentru activități administrative, deci și o posibilitate de a interacționa cu baza de date într-un mod facil.

Pentru a putea folosi aceasta interfață este necesar ca noi să creăm un user cu drepturi depline. În acest sens vă rog să aplicați comanda

**manage.py createsuperuser**

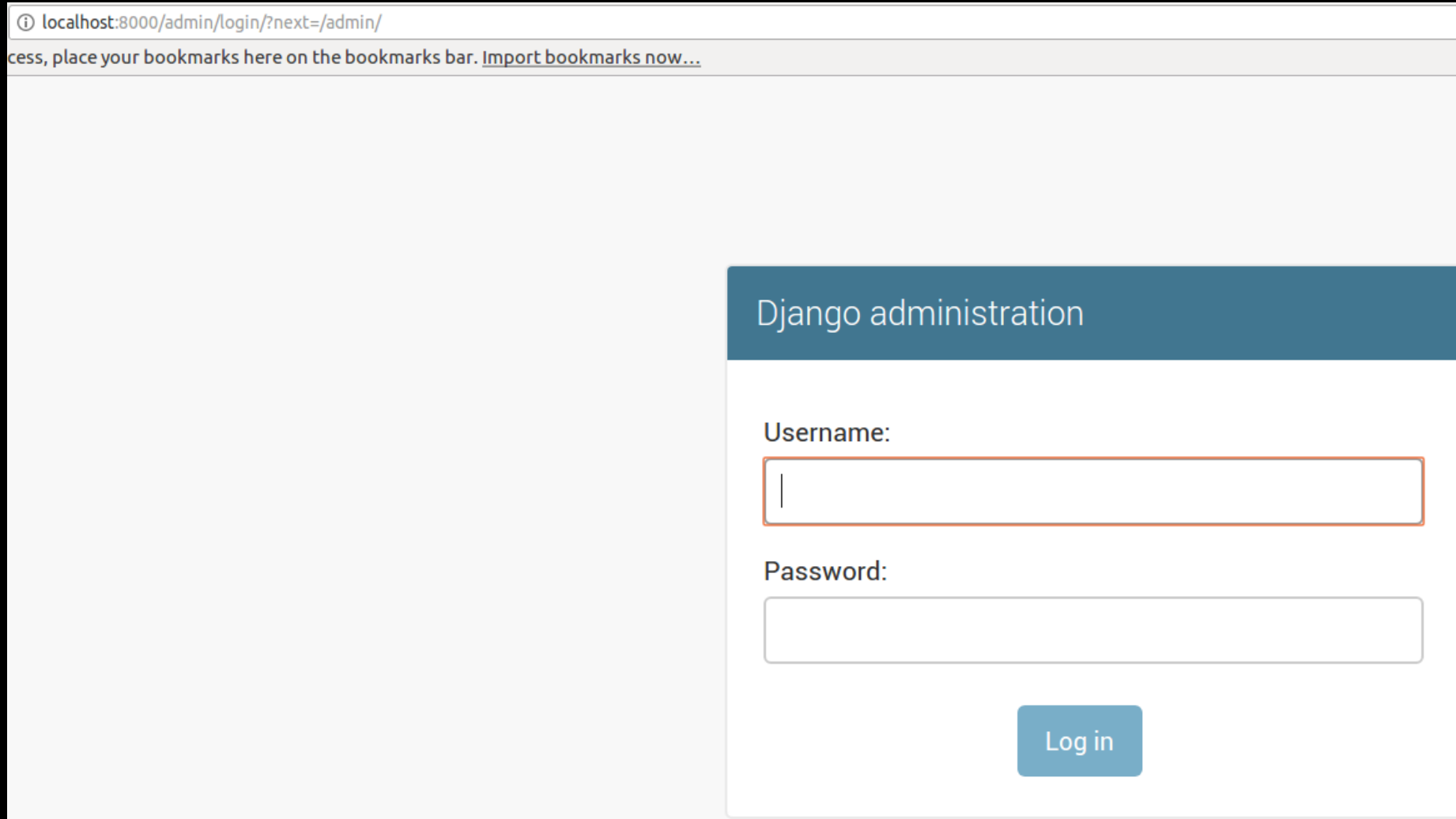
După aplicarea acestei comenzi veți fi întrebat ce user și parola doriți pentru contul de superuser. Parola trebuie să fie de 8 caractere combinate.

```
\catalog>manage.py createsuperuser
Username (leave blank to use 'administrator'): admin
Email address: catalin.popescu@bitacademy.net
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Password:
Password (again):
Error: Your passwords didn't match.
Password:
Password (again):
Superuser created successfully.
```



# ADMIN.PY

Dacă pornim serverul prin manage.py runserver apoi accesăm <http://localhost:8000/admin/> atunci putem vedea pagina administrativă



The screenshot shows a web browser window with the address bar displaying `localhost:8000/admin/login/?next=/admin/`. Below the address bar is a bookmark bar with the text "cess, place your bookmarks here on the bookmarks bar. [Import bookmarks now...](#)". The main content area is white and contains a login form titled "Django administration" in a blue header. The form has two input fields: "Username:" and "Password:". Below the password field is a blue "Log in" button.

localhost:8000/admin/login/?next=/admin/

cess, place your bookmarks here on the bookmarks bar. [Import bookmarks now...](#)

Django administration

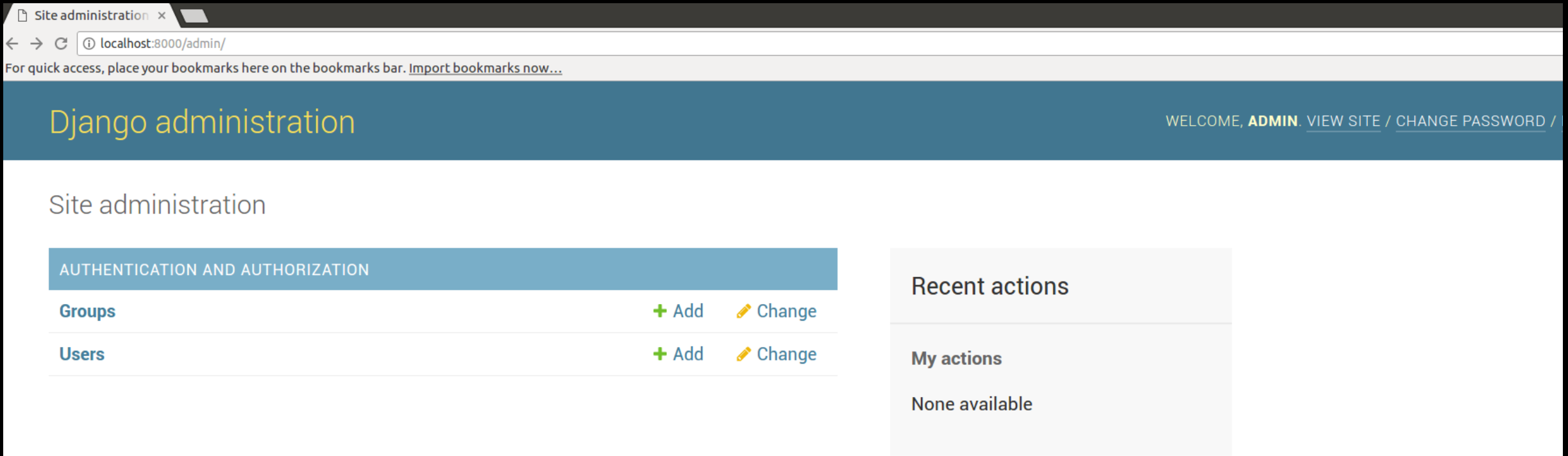
Username:

Password:

Log in



Din păcate, în pagina administrativă avem default acces doar la o categorie numită Users.



The screenshot shows the Django administration interface in a web browser. The browser tab is labeled 'Site administration' and the address bar shows 'localhost:8000/admin/'. Below the browser window, the Django admin page is displayed. The header 'Django administration' is on the left, and 'WELCOME, ADMIN. [VIEW SITE](#) / [CHANGE PASSWORD](#)' is on the right. The main content area is titled 'Site administration' and contains a table with two rows: 'Groups' and 'Users'. Each row has '+ Add' and 'Change' (with a pencil icon) links. To the right of the table, there is a 'Recent actions' section with the heading 'My actions' and the text 'None available'.

AUTHENTICATION AND AUTHORIZATION	
Groups	<a href="#">+ Add</a> <a href="#">Change</a>
Users	<a href="#">+ Add</a> <a href="#">Change</a>

### Recent actions

#### My actions

None available

Dacă ne dorim sa administrăm în acest GUI și tabelul Student atunci trebuie să realizăm anumite modificări. În acest sens accesăm fișierul main/admin.py unde adaugăm următoarele linii:

```
from main.models import Student
```

```
# Register your models here.  
class studentAdmin(admin.ModelAdmin):  
    list_display=["nume", "email", "proiect", "nota"]  
  
admin.site.register(Student, studentAdmin)
```

# ADMIN.PY



Sa adaugăm câteva intrări unde nota este 10, cel puțin o intrare cu o nota diferita de 10 și alte intrări unde nota este cea default (nu setați nota)

Django administration WELCOME, **ADMIN** [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

[Home](#) > [Main](#) > [Students](#) > Add student

### Add student

**Name:**

**Prenume:**

**Description:**

**Proiect:**  No file chosen

**Email:**

**Nota:**

Pentru a putea vedea aceste date introduse din baza de date prin Django Putem folosi shell-ul django. După rularea comenzii **manage.py shell** se accesează o secțiune care permite apelarea similar ca în shell-ul IDLE python.

Aici dăm comanda `from main.models import Student`. Astfel din aplicație din fișierul `models.py` avem clasa `Student` pe care o importam.

```
In [1]: from main.models import Student
```

```
In [2]: print Student.objects.all()
```

```
<QuerySet [<Student: Student object>, <Student: Student object>,  
<Student: Student object>, <Student: Student object>, <Student:  
Student object>]>
```

Dacă numărul de obiecte este mare devine mai facil sa ai o metodă care permite filtrarea obiectelor după o anumita valoare.

```
In [4]: for obj in Student.objects.all():  
        ....:     print obj.numa,obj.nota  
        ....:  
Ion 0  
Maria 0  
Andrei 10  
Petre 10  
George 8
```

Mai jos putem vedea utilizarea metodei `get()` care permite filtrarea unei intrări după valoarea deținută.

```
In [5]: un_student=Student.objects.get(nota=8)
```

```
In [6]: un_student
```

```
Out[6]: <Student: Student object>
```

```
In [7]: un_student.nume,un_student.proiect
```

```
Out[7]: (u'George', <FieldFile: proiecte/python_logo4..png>)
```

```
In [8]:
```

---

Metoda `get()` din păcate întâmpina probleme atunci când se returnează un număr mai mare sau egal cu doi de obiecte sau nici un obiect.

```
Command Prompt - manage.py shell

In [8]: un_student=Student.objects.get(nota=7)
-----
-----
DoesNotExist                                Traceback (most recent
call last)

    377         raise self.model.DoesNotExist(
    378             "%s matching query does not exist." %
--> 379             self.model._meta.object_name
    380         )
    381         raise self.model.MultipleObjectsReturned(

DoesNotExist: Student matching query does not exist.
```



Metoda `get()` din păcate întâmpina probleme atunci când se returnează un număr mai mare sau egal cu doi de obiecte sau nici un obiect.

```
In [9]: un_student=Student.objects.get(nota=10)
-----
-----
MultipleObjectsReturned                                Traceback (most recent
  call last)
    381         raise self.model.MultipleObjectsReturned(
    382             "get() returned more than one %s -- it retur
ned %s!" %
--> 383             (self.model._meta.object_name, num)
    384         )
    385
MultipleObjectsReturned: get() returned more than one Student --
it returned 2!
```

Soluția la aceasta problema este metoda `filter()`. Aceasta permite extragerea informației într-o listă. Dacă criteriul oferit returnează intrări duplicat atunci lista va fi mai mare, dacă nu returnează nimic atunci lista este goală. Lista respectiva poarta numele de `QuerySet`.

```
In [10]: un_student=Student.objects.filter(nota=10)

In [11]: un_student
Out[11]: <QuerySet [<Student: Student object>, <Student: Student
object>]>

In [12]: un_student=Student.objects.filter(nota=7)

In [13]: un_student
Out[13]: <QuerySet []>
```

Ultima metoda studiata este `exclude()`, care permite extragerea informației excluzând rândurile unde criteriul oferit se potrivește.

```
In [14]: un_student=Student.objects.exclude(nota=10)

In [15]: print un_student
<QuerySet [<Student: Student object>, <Student: Student object>,
<Student: Student object>]>

In [16]: for obj in un_student:
.....:     print obj.num, obj.nota
.....:
Ion 0
Maria 0
George 8
```

Sa presupunem ca dorim ca obiectele să fie sortate după nota. Atunci putem aplica metoda `order_by(criteriu)` la un `QuerySet`. Metoda `order_by` funcționează la orice `QuerySet` deci implicit și la `exclude()` sau `filter()`. Dacă ne dorim un exemplu de utilizare a metodei `order_by()` atunci iată un exemplu mai jos.

```
In [5]: Student.objects.all()
Out[5]: <QuerySet [<Student: Student object>, <Student: Student
object>, <Student: Student object>, <Student: Student object>, <
Student: Student object>]>
```

```
In [6]: for obj in Student.objects.all().order_by('nota'):
...:     print obj.nume,obj.nota
...:
```

Ion 0

Maria 0

George 8

Andrei 10

Petre 10

Fișierul `urls.py` se ocupă de url routing cu rolul de a accesa diferite pagini web în funcție de linkul apelat. În acest sens trebuie să revizuiți și să adăugați câteva elemente din regular expression. Aceste expresii regulate se aplică URL-ului după partea de bază.

Se introduce simbolul special `^` care indică ca elementul care urmează trebuie să fie la început de șir de caractere. Spre exemplu dacă avem tiparul `^ceva` atunci se va mapa cu `ceva` dar nu se va mapa cu `altceva`.

```
>>> import re
>>> x=re.search("^ceva", "altceva")
>>> print x
None
>>> x=re.search("^ceva", "ceva ce imi doresc")
>>> print x.group(0)
ceva
```

Un alt element introdus aici este \$ care are însemnătatea ca acel element trebuie să fie la finalul șirului de caractere, la polul opus simbolului ^ ca și comportament.

De reținut ca tiparul '^\$' va mapa exclusiv cu un sir gol.

```
>>> x=re.search("ceva$", "altceva")
>>> print x.group(0)
ceva
>>> x=re.search("ceva$", "ceva ce imi doresc")
>>> print x
None
```

# URLS



În proiect\urls.py trebuie sa adaugăm diferite linii pentru a atinge comportamente diferite în funcție de url.

În partea superioara trebuie sa adaugăm linia:

**from main import views**

În lista urlpatterns trebuie sa adaugam doua intrari. Fiecare intrare adaugata este formata dintr-un obiect al clasei url care are 3 elemente în interior:

`url(r'^$', views.index, name='index'),`

URL(tipar după care se face url routing,

Functia/clasa apelata din views.py în acest caz,

Cum se numește intrare url pt o indicare inversa)



# URLS

A doua intrare ce trebuie adăugată în lista urlpatterns este:

`url(r'^entry/(\w+)', views.entry_detail,name='entry_detail')`

Astfel urls.py trebuie sa arate ca în figura de mai jos.

```
from django.conf.urls import url
from django.contrib import admin
from main import views

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', views.index,name='index'),
    url(r'^entry/(\w+)', views.entry_detail,name='entry_detail'),
]
```

# VIEWS



În views importam clasa HttpResponse din django.http:

```
from django.http import HttpResponse
```

Scopul clasei HttpResponse este sa returnam în mod direct un răspuns pentru un url pentru functia index și functia entry\_detail.

Prima funcție numita index primește un parametru de intrare numit request ce va fi populat în mod automat. Acesta stocheaza informații despre conexiune. Vom returna un obiect al clasei HttpResponse cu un sir de caractere inclus într-un tag de tip p

```
def index(request):
```

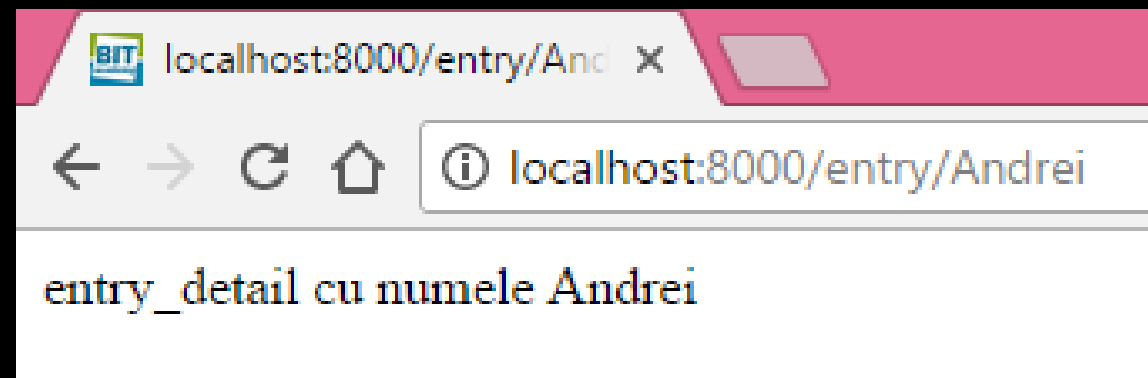
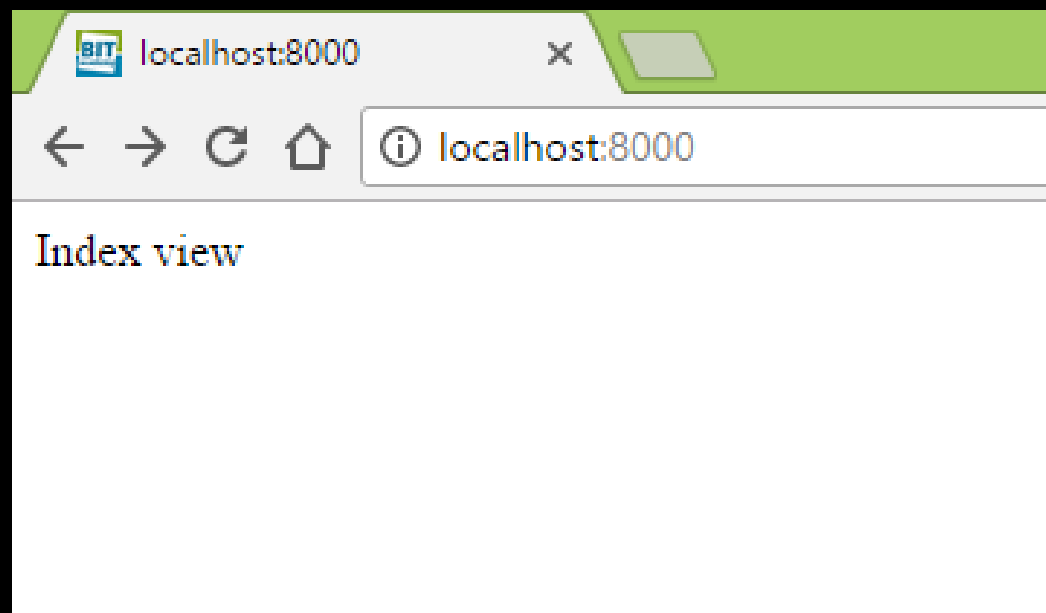
```
    return HttpResponse('<p>Index view</p>')
```

# VIEWS

Acum trebuie sa rulam serverul, deci aplicați comanda `manage.py runserver`

La apelarea `http://localhost:8000/` se poate vedea o pagina în care apare textul Index view

La apelarea `http://localhost:8000/entry/Andrei` se poate vedea o pagina în care apare textul `entry_detail` cu numele Andrei



Într-o pagină web multe din informații sunt duplicat. Astfel se introduce conceptul de template. Un template permite scalabilitate în sensul că se va adăuga posibilitatea ca pagina aleasa de url să conțină informații formate corespunzător dintr-o pagina de bază. Vom discuta de asta mai târziu. Pana atunci trebuie atinse anumite elemente.

În fișierul views.py trebuie adăugate anumite importuri deoarece se dorește eliminarea unei metode directe de a returna pagina și se dorește returnarea valorilor din baza de date prin constrângere după criteriile impuse utilizand filter() sau exclude(). Astfel se vor adăuga următoarele linii:

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

```
from main.models import Student
```

Primul import îl utilizam pentru a informa utilizatorul ca pagina nu exista. Al doilea import l-am văzut prima oară în shell-ul django când am apelat manage.py shell în vederea utilizării metodelor exclude() sau filter().

# TEMPLATE



Primul import îl utilizam pentru a informa utilizatorul ca pagina nu exista. Al doilea import l-am văzut prima oară în shell-ul django când am apelat `manage.py shell` în vederea utilizării metodelor `exclude()` sau `filter()`.

```
def index(request):
```

```
    intrari=Student.objects.all()
```

```
    return render(request,'index.html',{'entryS':intrari})
```

Functia `index` salvează în variabila locala `intrari` un `QuerySet` returnat de metoda `all()`. Se returnează o apelare a funcției `render` care are trei parametri:

- Primul parametru este `request` prin care se manipulează conexiunea.
- Al doilea parametru este template-ul utilizat. Astfel noi împingem informație către un template care va afișa informația într-un mod convenabil.
- Al treilea parametru reprezintă ceva ce poartă numele de template context. Acest parametru este un dicționar prin care trimitem date locale acelei funcții către template-ul indicat. În cazul funcției `index` luam variabila locală `intrari` și o facem disponibilă în cadrul template-ului ca fiind `entryS`

# TEMPLATE



În corpul funcției `entry_detail` vedem extragerea prin `get()` a informației primite ca și parametru de intrare. Dacă `get` nu găsește o intrare cu id-ul respectiv atunci va returna eroare. Atunci se folosește `except` și va returna o pagina 404, în caz contrar returnăm o apelare a funcției `render` care indica utilizarea template-ului `entry_detail.html` și transmiterea datelor extrase prin `get()`.

```
def entry_detail(request,id):
```

```
    try:
```

```
        myID=Student.objects.filter(id=id)
```

```
    except:
```

```
        raise Http404("Numele nu exista")
```

```
    else:
```

```
        return render(request,'entry_detail.html',{'myID':myID})
```



# TEMPLATE



Primul import îl utilizam pentru a informa utilizatorul ca pagina nu exista. Al doilea import l-am văzut prima oară în shell-ul django când am apelat `manage.py shell` în vederea utilizării metodelor `exclude()` sau `filter()`.

```
def index(request):
```

```
    intrari=Student.objects.all()
```

```
    return render(request,'index.html',{'entryS':intrari})
```

Functia `index` salvează în variabila locala `intrari` un `QuerySet` returnat de metoda `all()`. Se returnează o apelare a funcției `render` care are trei parametri:

- Primul parametru este `request` prin care se manipulează conexiunea.
- Al doilea parametru este template-ul utilizat. Astfel noi împingem informație către un template care va afișa informația într-un mod convenabil.
- Al treilea parametru reprezintă ceva ce poartă numele de template context. Acest parametru este un dicționar prin care trimitem date locale acelei funcții către template-ul indicat. În cazul funcției `index` luam variabila locală `intrari` și o facem disponibilă în cadrul template-ului ca fiind `entryS`



## Cum arata views.py

```
from django.http import Http404
from main.models import Student

def index(request):
    intrari=Student.objects.all()
    return render(request, 'index.html', {'entryS':intrari})

def entry_detail(request, id):
    try:
        myID=Student.objects.get(id=id)
    except:
        raise Http404("Numele nu exista")

    else:
        return render(request, 'entry_detail.html', {'myID':myID})
```

# TEMPLATE



Deci creați directorul templates în directorul unde se regăsește fișierul manage.py

În interiorul fișierului index.html din directorul templates adăugați linia:

```
<h2> Acasă </h2>
```

În interiorul fișierului entry\_detail.html din directorul templates adăugați linia:

```
<h2> Entry detail </h2>
```

# TEMPLATE

În cele ce urmează trebuie să informăm django că exista un director cu aceste template-uri. Prin urmare, trebuie sa adaugam în setting.py directorul templates. Astfel ca în settings.py exista lista stocata sub variabila TEMPLATES care conține în interior la rândul 57 DIRS ce stocheaza o lista. Vă rog sa adăugați acolo „templates”

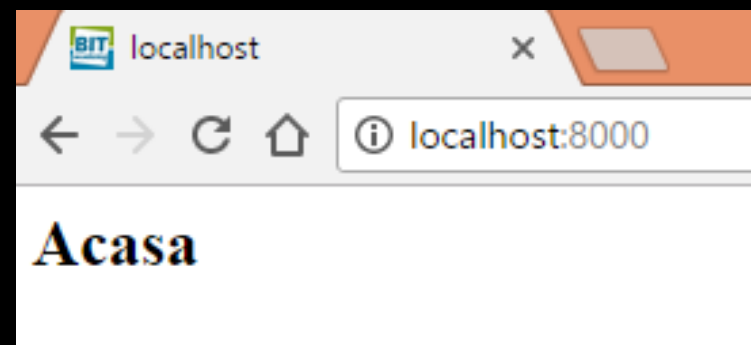
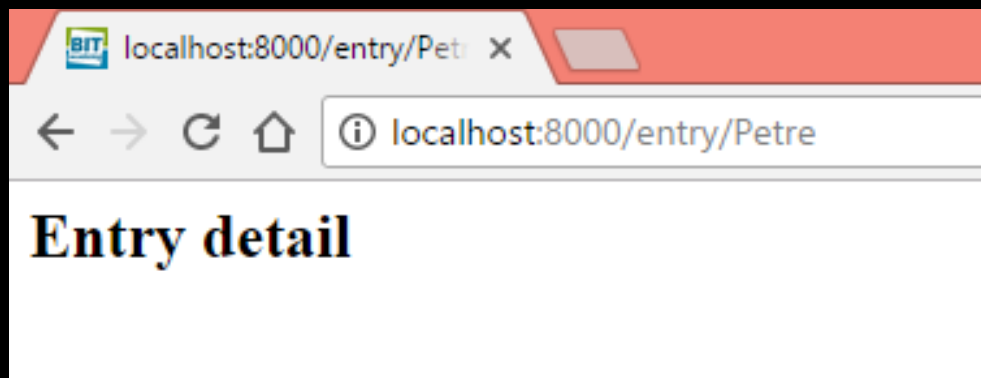
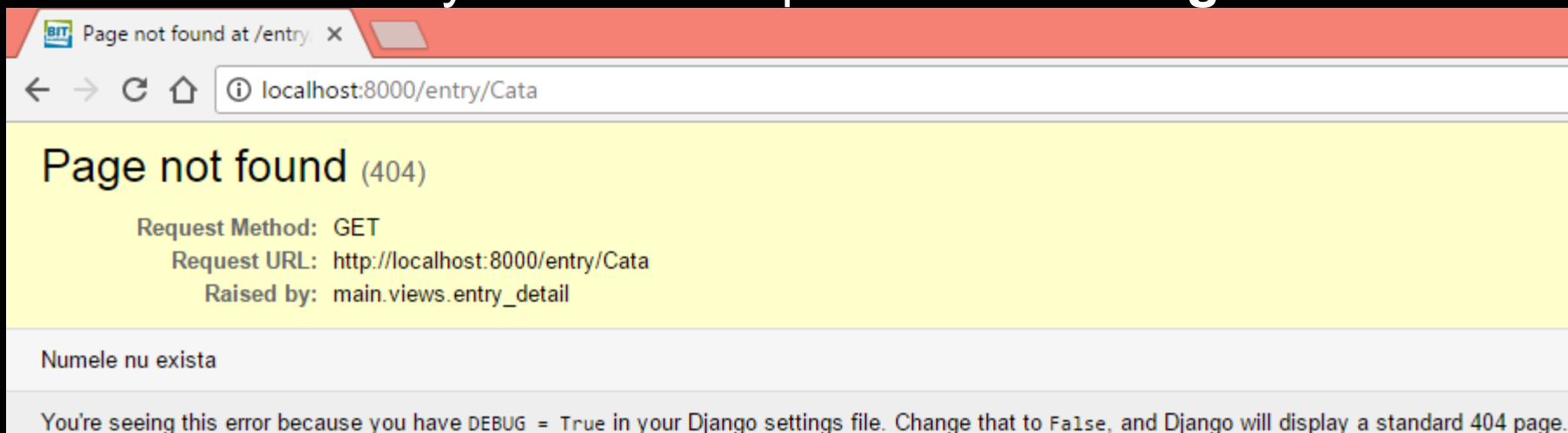
```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': ['templates'],  
        'APP_DIRS': True,  
        'OPTIONS': {
```

# TEMPLATE

La apelarea `http://localhost:8000/` se poate vedea textul Acasa

La apelarea `http://localhost:8000/entry/Petre` atunci se poate vedea Entry detail

Dacă în schimb apelăm o intrare care nu exista în baza de date în link cum ar fi `http://localhost:8000/entry/Cata` atunci putem vedea **Page not found**.



# DJANGO TEMPLATE LANGUAGE



Django Template Language reprezintă un set de reguli care permite manipularea elementelor care ai fost transmise de la views.py

Template reprezintă un fisier text care permite crearea de cod HTML, XML etc. Un template poate fi format din variabile și tag-uri.

Prima regula pe care o studiem este afisarea unei variabile primite din views.py

Acest lucru se face prin `{{ variabila }}`.

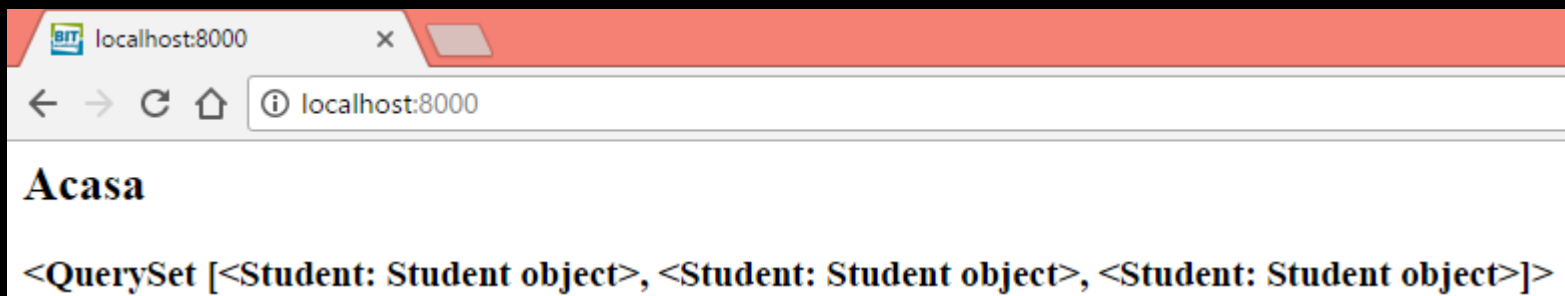
Pentru a înțelege cât mai bine cum se folosește vă rog sa adăugați la finalul fișierului index.html linia:

```
<h3> {{entryS}} </h3>
```

# DJANGO TEMPLATE LANGUAGE



La apelarea `http://localhost:8000/` se poate vedea textul Acasă, iar pe rândul următor un query set





# DJANGO TEMPLATE LANGUAGE



La o variabila de tip sir de caractere se poate utiliza pipe în vederea prelucrării automate a informației. Conceptul poarta numele de filter. Construcția este :

**{{entryS | filter}}**

Din cele mai utilizate filtre prezentam:

- lower - sirul devine lower
- upper - sirul devine upper
- center:"15" – sirul este centrat la 15 caractere
- capfirst - mărește primul caracter al sirului de caractere
- cut:" " - elimina ce caractere se regasesc intre sirul de caractere. (replace cu un sir vid din python)
- striptags – încearcă eliminarea tag-urilor HTML
- escape – convertește cod html în simboluri vizibile pentru html. Exemplu < este convertit in &lt;

# DJANGO TEMPLATE LANGUAGE



Parametrul primit de la views.py poate fi o lista deci avem nevoie de un filtru care permite împărțirea în doua secțiuni returnând prima parte. Structura este

```
{{ lista|slice:"2" }}
```

Deci dacă avem o lista [1,2,3,4] dacă aplicam slice:2 atunci va returna [1,2]

Un alt filtru care poate fi aplicat la o lista este length. Acesta returnează lungimea listei

Filtrul length\_is testează dacă lungimea secvenței (sir, lista etc) este de o anumită dimensiune returnând un Boolean. Exemplu:

```
{{ valoare | length_is:"4" }}
```

**Mai multe exemple aici.**

# DJANGO TEMPLATE LANGUAGE



O a doua categorie de constructii DTL pe care le găsim în template este tag-ul.

Tag-urile pot fi de mai multe feluri. Toate au în comun faptul ca încep cu {% construcție %} și se termina cu {% endconstrucție %}

Primul tip se numește for. Acesta permite afișarea unui tabel sau alte task-uri ciclice.

```
{% for element_din_lista in {{lista}} %}
```

```
<dif> element_din_lista
```

```
</dif>
```

```
{% endfor %}
```

# DJANGO TEMPLATE LANGUAGE



Un alt tip de tag este if, else si elif. Structura pentru if-else este :

```
{%if 2 != 0 %}
```

```
<p>ceva</p>
```

```
{% else %}
```

```
<p>altceva</p>
```

```
{% endif %}
```

Structura pentru if-elif-else este :

```
{% if var1 %}
```

```
<p>{{ var1 }}</p>
```

```
{% elif var2 %}
```

```
<p>{{ var2 }}</p>
```

```
{% elif var3 %}
```

```
<p>{{ var3 }}</p>
```

```
{% else %}
```

```
<p>Nu am gasit!</p>
```

```
{% endif %}
```

Template-ul index.html devine:

```
<h2> Acasa </h2>
```

```
<h3> Lista de studenti </h3>
```

```
<ul>
```

```
  {% for student in entryS %}
```

```
    <li>
```

```
      <a href="{% url 'entry_detail' student.id %}">{{student.email}}</a>
```

```
    </li>
```

```
  {% endfor %}
```

```
</ul>
```

Regăsim în interior un {% for %} care se termina cu {% endfor %}. În interiorul construcției {% for %} avem variabila student care devine un obiect al QuerySet-ului.

Dacă în urls.py avem numele url-ului test atunci trebuie updatat și în template:

```
<h2> Acasa </h2>
<h3> Lista de studenti </h3>
<ul>
{% for student in entryS %}
  <li>
    <a href="{% url 'test' student.id %}">{{student.email}}</a>
  </li>
{% endfor %}
</ul>
```

Tot aici putem vedea construcția `<a href="{% url 'entry_detail' student.id %}">{{student.email}}</a>`. Aceasta are rolul de a adauga în pagina URL-uri.

Conform ultimei linii din `urls.py` avem: `url(r'^entry/(\w+)', views.entry_detail, name='entry_detail')`

Asta se traduce cu accesarea paginii `http://localhost:8000/entry/`

Când noi folosim url `'entry_detail'` atunci folosim numele acelui url pentru redirectare de trafic.

Dacă în `urls.py` avem numele url-ului test atunci trebuie updatat și în template:

```
urlpatterns = [  
    url(r'^admin/', admin.site.urls),  
    url(r'^$', views.index, name='index'),  
    url(r'^entry/(\w+)', views.entry_detail, name='test'),  
]
```



```
<a href="{% url 'entry_detail' student.id %}">{{student.email}}</a>.
```

Fiecare intrare din baza de date este indexata după un id. Astfel fiecare intrare are un singur id și un id indica o singura intrare. Astfel dacă noi trimitem acel id către pagina detail am putea face referire la intrarea respectiva în vederea extragerii de informație.

Sintaxa **{{student.email}}** este încadrată între cele doua tag-uri `<a></a>` deci va fi textul afișat în pagina. Dacă dorim să afișăm numele fiecărui utilizator în pagina putem schimba cu **{{student.num}}**

```
<h2> Acasa </h2>
<h3> Lista de studenti </h3>
<ul>
{% for student in entryS %}
  <li>
    <a href ="{% url 'entry_detail' student.id %}">{{student.num}}</a>
  </li>
{% endfor %}
</ul>
```

File Edit View History Tools People Help

localhost x

localhost:8000

## Acasa

### Lista de studenti

- [Ion](#)
- [Maria](#)
- [Andrei](#)
- [Petre](#)
- [George](#)

```
<a href="{% url 'index' %}">Inapoi la lista de studenti</a>
<table>

    <h2>Student situatie </h2>
    <tr><td> Nume:</td> <td>{{myID.num}} </td></tr>
    <tr><td> Prenume:</td> <td>{{myID.prenume}}</td> </tr>
    <tr><td> Email:</td> <td>{{myID.email}}</td> </tr>

    {%if myID.nota != 0 %}
        <tr><td> Nota:</td> <td>{{myID.nota}}</td> </tr>
    {% else %}|
        <tr><td> Nota:</td><td> Proiectul nu este notat. Va rugam sa asteptati!</td> </tr>
    {% endif %}

    <tr><td> Detalii:</td> <td>{{myID.description}}</td></tr>
    <tr><td> Proiect:</td> <td>{{myID.proiect|cut:"proiecte/"}}</td></tr>

</table>
```

Aici regăsim în partea superioara un tag <a> pentru a ne întoarce înapoi la index.

Din clasa entry\_detail din views.py primim myID care conține un obiect de tip QuerySet. Astfel ca putem extrage informația apelând variabila {{myID.nume}}, Tot aici regăsim și o construcție if-else care verifica dacă nota este diferita de 0. Dacă este 0 atunci afișează proiectul nu este notat. În caz contrar, returnează nota.

La ultima variabila regăsim un filtru cut: {{myID.proiect|cut:"proiecte/"}}.

Variabila myID.project returnează numele proiectului în formatul următor:

**proiecte/NUMEPROIECT**

Unde NUMEPROIECT este numele fișierului atașat. Se dorește eliminarea **proiecte/** prin urmare folosim filtrul cut.

Django aplica conceptul DRY (Don't Repeat Yourself) se dorește formatarea paginii într-un mod ierarhic

Exista posibilitatea ca o moștenire să se realizeze la nivel de template. Astfel exista o pagina de baza, un template, care să fie moștenit de toate celelalte pagini.

Pentru ca acest concept să fie implementat trebuie create secțiuni de tip block adică

```
{ % block content % }
```

```
{ % endblock content % }
```

unde cuvântul content este interschimbabil cu alt cuvânt.

O astfel de implementare are doua părți:

- Setarea unui template base.html.
- Moștenirea acestui template de restul de template-uri.

# BLOCK TAG



Mai jos se poate vedea ca trebuie sa scriem în base.html

```
<!DOCTYPE html>
<html> <head>
    <title>Catalog</title>
</head>
<body>
    <h1 align="center">Catalog</h1>

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

# BLOCK TAG



A doua secțiune a unui block tag este moștenirea. Astfel unde dorim să moștenim această pagină, adică în templates, se va adăuga:

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

Iată cum trebuie să arate index.html acum:

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

```
{% block content %}
```

```
<h3> Lista de studenti </h3>
```

```
<ul>
```

```
{% for student in entryS %}
```

```
  <li>  <a href="{% url 'entry_detail' student.id %}">{{student.email}}</a>
```

```
  </li>
```

```
{% endfor %}
```

```
</ul>
```

```
{% endblock content %}
```



# BLOCK TAG



entry\_detail.html va fi modificat identic cum e mai jos:

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

```
{% block content %}
```

```
<h3> Lista de studenti </h3>
```

```
<ul>
```

```
{% for student in entryS %}
```

```
  <li>  <a href="{% url 'entry_detail' student.id %}">{{student.email}}</a>
```

```
  </li>
```

```
{% endfor %}
```

```
</ul>
```

```
{% endblock content %}
```

# CONTACT



Dorim în cele ce urmează să introducem și o zonă de contact a acestei pagini. În acest sens trebuie să creăm o nouă funcționalitate.

Prin urmare trebuie să adăugăm în `urls.py` următoarea linie:

```
url(r'^contact/$', views.contact, name='contact'),
```

Această linie are rolul de a redirecta către funcția `contact` din `views.py` traficul care accesează `http://localhost:8000/contact`

În `views.py` trebuie introdusă o funcție numită `contact`:

```
def contact(request):  
    return render(request, 'contact.html', {})
```

# CONTACT



Ultimul pas reprezintă setarea unui template numit contact.html

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

```
{% block content %}
```

```
<p>Contact: Popescu Catalin</p>
```

```
<p>Email:office@site.com</p>
```

```
<p>Tel:070.111.111</p>
```

```
{% endblock %}
```

În scopul de a accesa contact introducem în pagina de baza base.html un meniu

```
<!DOCTYPE html>
```

```
<html> <head>
```

```
    <title>Catalog</title>
```

```
</head>
```

```
<body>
```

```
    <h1 align="center">Catalog</h1>
```

```
    <div id="tabs">
```

```
        <ul>
```

```
            <li><a href="/">Acasa</a></li>
```

```
            <li><a href="/contact/">Contact</a></li>
```

```
        </ul>
```

```
    </div>
```

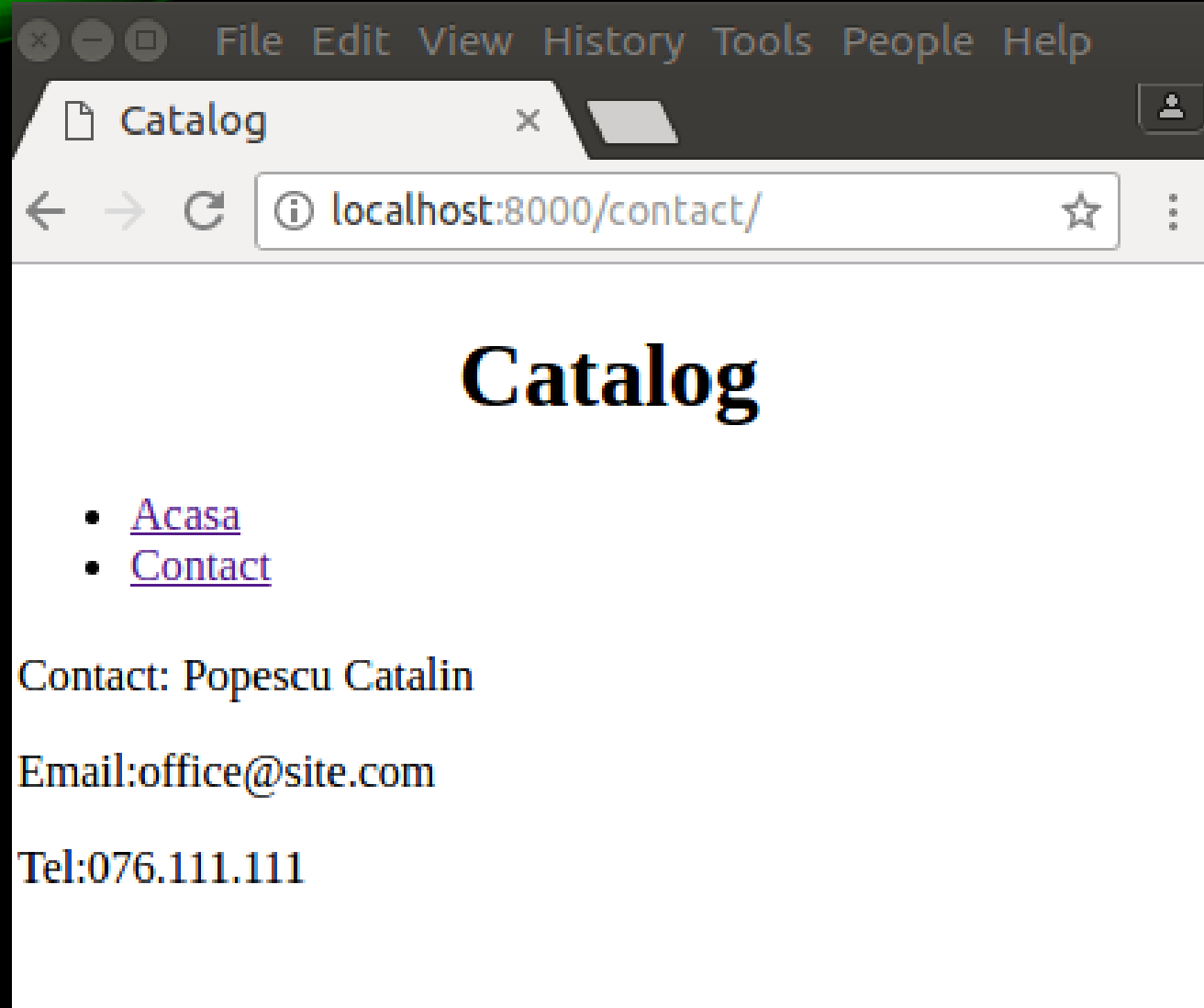
```
    {% block content%}
```

```
    {% endblock content %}
```

```
</body>
```

```
</html>
```

# MENIU



În scopul de a accesa contact introducem în pagina de baza base.html un meniu

```
<!DOCTYPE html>
```

```
<html> <head>
```

```
    <title>Catalog</title>
```

```
</head>
```

```
<body>
```

```
    <h1 align="center">Catalog</h1>
```

```
    <div id="tabs">
```

```
        <ul>
```

```
            <li><a href="/">Acasa</a></li>
```

```
            <li><a href="/contact/">Contact</a></li>
```

```
        </ul>
```

```
    </div>
```

```
    {% block content%}
```

```
    {% endblock content %}
```

```
</body>
```

```
</html>
```

# ADAUGAREA CSS ȘI JS



Vom adauga un fisier CSS si JS fără a discuta despre acest aspect. Aceste fișiere nu fac parte din cursul Python, dar nici nu fac nota discordanta fata de paginile web fiind tehnologii des întâlnite în conjuncție cu Django.

În acest sens se va adauga la finalul fisierului settings.py variabila `STATICFILES_DIRS` astfel:

```
STATICFILES_DIRS = (os.path.join(BASE_DIR), 'static')
```

Aceasta linie informează django ca în directorul **static** adăugat în directorul proiectului se pot adăuga fișiere JS sau CSS.

Vă rugăm să setati directorul static în directorul unde exista manage.py. În acest director se vor adăuga cele 4 fișiere ce se găsesc pe site: arhiva `fisiere_django_proiect_static.zip`



# ADAUGAREA CSS ȘI JS



Arhiva conține 4 fișiere: fundal.jpg, favicon.ico, main.css și main.js.

Fișierul favicon.ico se va solicita automat de către browser și nu trebuie să facem nimic pentru a activa aceasta utilizare.

Fișierul fundal.jpg este integrat în main.css, deci nu trebuie să facem nimic pentru a activa acest fundal.

În schimb trebuie să setăm main.css și main.js.

Pentru a fi aplicate fiecărei pagini web asociată site-ului trebuie să modificăm base.html.

Astfel în partea superioară se va insera {% load staticfiles %}

Această linie are rolul de a încărca fișiere statice indicate de settings.py.

# ADAUGAREA CSS ȘI JS



Astfel la finalul tagului head adică înainte de `</head>` adaugam:

```
<link rel="stylesheet" href="{% static 'main.css' %}">
```

```
</head>
```

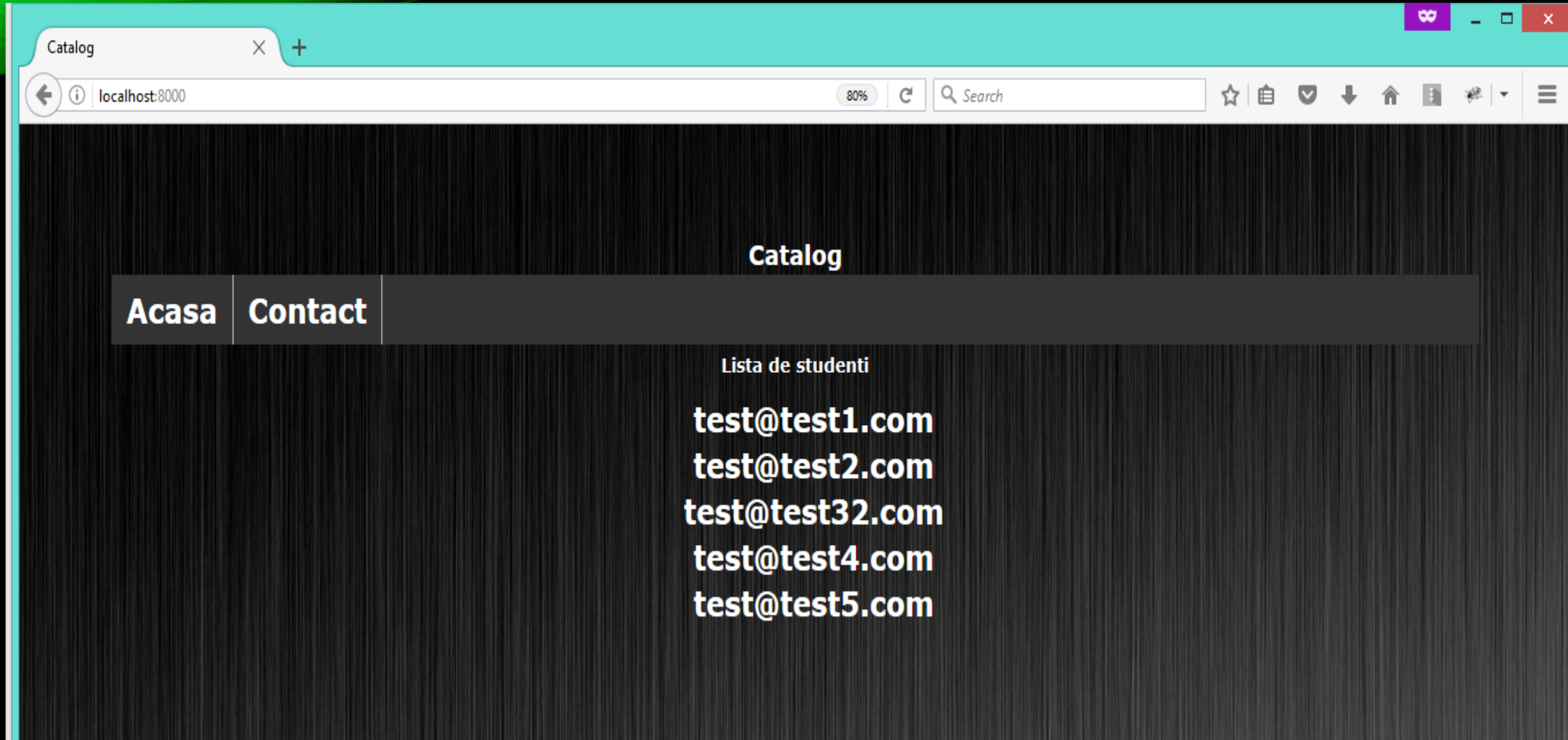
Aceasta linie indica ce fișier css va fi utilizat aici: main.css

Înainte de închiderea tag-ului body trebuie sa indicam fișierul js ce va fi utilizat. Construcția de mai jos are acest scop.

```
<script src="{% static 'main.js' %}"></script>
```

```
</body>
```

# ADAUGAREA CSS ȘI JS



VA MULTUMESC PENTRU PARTICIPARE

**La revedere!**