

Tecnologias e Programação Web

Resumos
2015/2016

João Alegria | 68661

9º Capítulo

Plataforma Django

Models

- MTV - *Model, Template, View*
- Model
 - Consiste na camada de acesso a dados - “Data Access Layer”
 - Esta camada permite definir, em relação aos dados:
 - acesso
 - validação
 - comportamento
 - relação entre os dados

Configuração da BD

- A definição do acesso aos dados, começa pela configuração do acesso à base de dados
- No ficheiro “settings.py”, procurar pela variável DATABASES e definir os parâmetros ENGINE, NAME, USER, PASSWORD, HOST, PORT.
 - Alguns destes parâmetros podem ser omitidos, conforme a BD a aceder.
- Por defeito, na criação do projeto é configurado o acesso à base de dados local SQLite. Para outras BDs, consultar a documentação: <https://docs.djangoproject.com/en/1.8/ref/settings/>

• Criação de um modelo:

- No ficheiro “models.py” da pasta “app”, definir as classes do modelo de dados a usar
- Para cada atributo das classes é instanciado um objeto do tipo “Field” e/ou subtipo, como: CharField, DateField, etc.
 - Documentação: <https://docs.djangoproject.com/en/1.8/ref/models/fields/#django.db.models.Field>
- Alguns atributos, representam a criação de relações entre as classes, tendo como efeito a criação de colunas como chaves estrangeiras (1:1, 1:M, M:1) ou tabelas de associação (M:N).
 - Documentação: <https://docs.djangoproject.com/en/1.8/ref/models/fields/#module-django.db.models.fields.related>

```
class Autor(models.Model):  
    nome = models.CharField(max_length=70)  
    email = models.EmailField()  
    def __str__(self):  
        return self.nome  
  
class Editor(models.Model):  
    nome = models.CharField(max_length=70)  
    cidade = models.CharField(max_length=50)  
    pais = models.CharField(max_length=50)  
    website = models.URLField()  
    def __str__(self):  
        return self.nome  
  
class Livro(models.Model):  
    titulo = models.CharField(max_length=100)  
    data_pub = models.DateField()  
    autores = models.ManyToManyField(Autor)  
    editor = models.ForeignKey(Editor)  
    def __str__(self):  
        return self.titulo
```

- **Relações entre as classes:**

- **1:M e M:1**

- Conseguida com um atributo da classe “models.ForeignKey”
 - Exemplo: Editor (1) : (M) Livro ou Livro (M) : (1) Editor
 - O atributo é colocado na classe Livro(M), aquela que representa muitos objetos para um.

- **1:1 (único)**

- Conseguida com um atributo da classe “models.OneToOneField”
 - Exemplo: Livro (1) : (1) Editor
 - O atributo “deve” ser colocado na classe que mais “necessita” da outra

- **M:N**

- Conseguida com um atributo da classe “models.ManyToManyField”
 - Exemplo: Livro (M) : (N) Autor
 - O atributo “deve” ser colocado na classe que mais “necessita” da outra
 - No presente exemplo, o autor pode existir, por si, sem necessidade de ter livros, mas o livro necessita de um autor

- A classe base “Model”, donde são derivadas todas as classes do modelo, possui todos os mecanismos necessários para interagir com a base de dados.
- Cada classe derivada é implementada na BD na forma de uma tabela e os seus atributos são implementados na forma de colunas (campos) na tabela.

- Exemplo da classe “Autor”

```
CREATE TABLE “app_autor” (
    “id” integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    “nome” varchar (70) NOT NULL,
    “email” varchar (254) NOT NULL
);
```

- Com vista à ativação do modelo, a aplicação web, “app”, deve ser incluída na variável INSTALLED_APPS do ficheiro “settings.py”, caso ainda não esteja.

- De seguida, deve-se proceder à validação do modelo (sintaxe e lógica) e para isso executar o seguinte comando na consola:

```
python manage.py check
```

- Dentro da pasta “app”, apaga-se a pasta “migrations”, caso exista, e executa-se os seguintes comandos, na consola:

```
python manage.py makemigrations app (produz código de migração)
python manage.py sqlmigrate app 0001 (opcional: mostra código SQL)
python manage.py migrate (produz as tabelas na BD)
```

- Se ainda não existir, poderá ser pedida a criação do superuser para a BD

```
python manage.py syncdb
```

- **Gestão dos Dados:**

- A plataforma Django possui um mecanismo que possibilita uma gestão muito facilitada de todos os dados pertencentes ao modelo de dados: o Django Admin Site
- A URL = <http://localhost:{port}/admin> dá acesso à área administrativa a qual permite, por defeito, gerir os utilizadores do site
- Nesta área, também é possível aceder e gerir os dados definidos no modelo

- Para ativar o Django Admin Site:

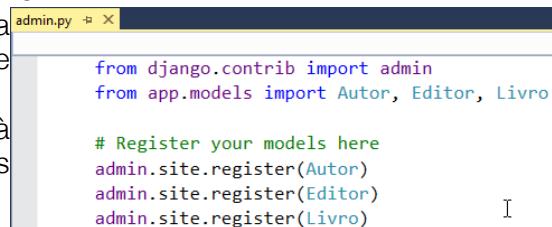
- Descomentar todas as linhas referentes à parte de admin nos ficheiros:
 - settings.py
 - urls.py

- Testar o Django Admin Site:

- Executar o projeto e aceder ao link: <http://localhost:{port}/admin>

- Adicionar o modelo de dados ao Admin Site

- Criar um ficheiro "admin.py" na pasta "app" e registar as classes que se pretendem gerir
- Executar de novo o projeto, voltar à página de administração e os dados tornam-se acessíveis



```
admin.py < X
from django.contrib import admin
from app.models import Autor, Editor, Livro

# Register your models here
admin.site.register(Autor)
admin.site.register(Editor)
admin.site.register(Livro)
```

- **Inserir um objeto**

```
a = Autor (nome='Antonio Pedro', email='apero@email.com')
a.save()
```

- **Selecionar todos os objetos**

```
Autor.objects.all()
```

- **Filtrar objetos**

```
Autor.objects.filter(nome='Autor1')
```

- **Filtrar por nome e por e-mail**

```
Autor.objects.filter(nome='Autor1', email='...')
```

- **Filtrar por um nome parecido**

```
Autor.objects.filter(nome_contains='Autor')
```

- **Aceder a um único objecto**

```
Autor.objects.get(email='autor1@email.com')
```

- **Ordenação**

```
Editor.objects.order_by("cidade", "pais")
```

- **Filtragem e Ordenação**

```
Editor.objects.filter(pais='Portugal').order_by("-cidade")
```

- **Selecionar o primeiro resultado**

```
Editor.objects.order_by("cidade", "pais")[0]
```

- **Inserir um objeto**

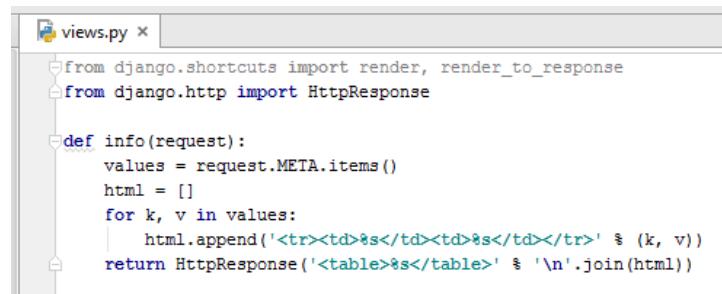
```
Autor.objects.get(email='autor1@email.com').delete()
```

Receção de Dados

- O objeto “request” do tipo “HttpRequest” permite aceder a um conjunto vasto de dados que são recebidos pelo web server
- Esses dados podem ser acedidos diretamente através de alguns atributos e métodos dedicados como:

```
request.path, request.get_host(), request.is_secure()
```

- Ou podem ser acedidos através do dicionário “request.META” que contém toda a informação presente no *header* do protocolo HTTP
- Exemplo de uma *view* para mostrar todos os dados presentes no *header* HTTP



```
views.py x
from django.shortcuts import render, render_to_response
from django.http import HttpResponseRedirect

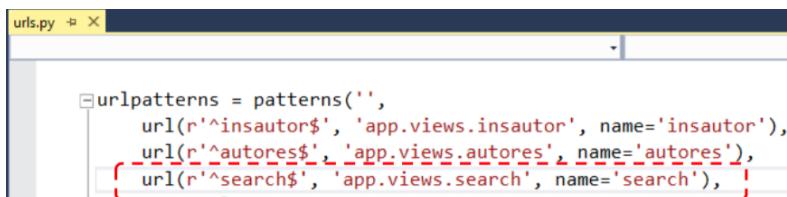
def info(request):
    values = request.META.items()
    html = []
    for k, v in values:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
    return HttpResponseRedirect('<table>%s</table>' % '\n'.join(html))
```

Forms

- Os *Forms* são os elementos HTML por excelência para o envio/receção de dados do cliente para o servidor.
- Do lado do browser (cliente) o *Form* pode usar o método Get ou o método Post, para enviar os dados nele contidos
- Do lado do servidor, a *view* invocada pode recorrer aos dicionários “request.GET” e “request.POST” para aceder aos dados recebidos

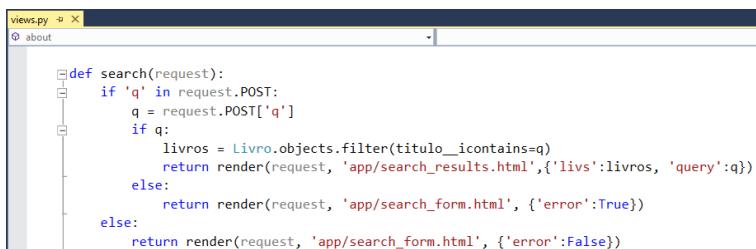
- Exemplo:

- Criação de um *Form* para pesquisar pelos títulos dos livros no modelo de dados anteriormente criado:



```
urls.py
urlpatterns = patterns('',
    url(r'^insautor$', 'app.views.insautor', name='insautor'),
    url(r'^autores$', 'app.views.autores', name='autores'),
    url(r'^search$', 'app.views.search', name='search'),
```

- Definição da *view*:



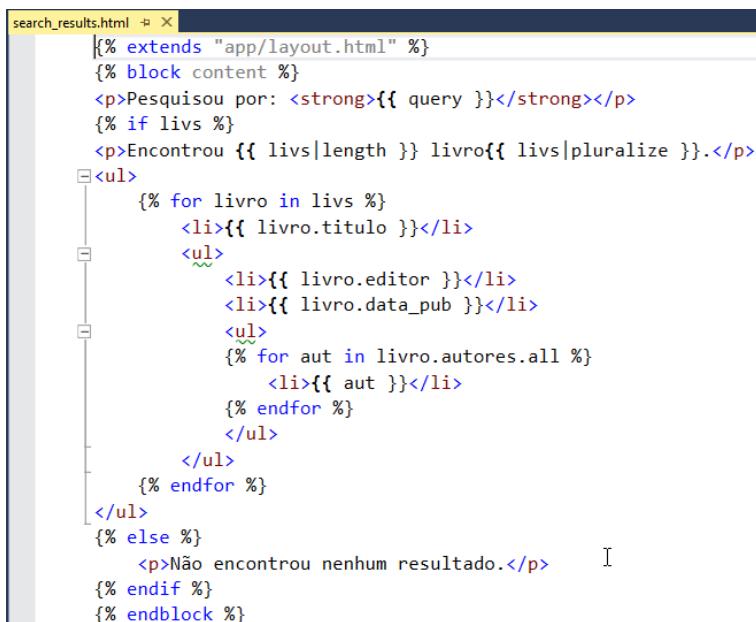
```
views.py
def search(request):
    if 'q' in request.POST:
        q = request.POST['q']
        if q:
            livros = Livro.objects.filter(titulo__icontains=q)
            return render(request, 'app/search_results.html', {'livs':livros, 'query':q})
        else:
            return render(request, 'app/search_form.html', {'error':True})
    else:
        return render(request, 'app/search_form.html', {'error':False})
```

- Definição da *template* para pesquisa:



```
search_form.html
{% extends "app/layout.html" %}
{% block content %}
{% if error %}
    <p style="color:red;">ERRO: Insira um termo a pesquisar.</p>
{% endif %}
<form action="/search" method="post">
    {% csrf_token %}
    <input type="text" name="q">
    <input type="submit" value="Search">
</form>
{% endblock %}
```

- Definição da *template* para os resultados:



```
search_results.html
{% extends "app/layout.html" %}
{% block content %}
<p>Pesquisou por: <strong>{{ query }}</strong></p>
{% if livs %}
<p>Encontrou {{ livs|length }} livro{{ livs|pluralize }}.</p>



    {% for livro in livs %}
        <li>{{ livro.titulo }}</li>
        <ul>
            <li>{{ livro.editor }}</li>
            <li>{{ livro.data_pub }}</li>
            <ul>
                {% for aut in livro.autores.all %}
                    <li>{{ aut }}</li>
                {% endfor %}
            </ul>
        </ul>
    {% endfor %}


{% else %}
    <p>Não encontrou nenhum resultado.</p>
{% endif %}
{% endblock %}
```

Persistência de Estado: Sessões e Autenticação

- Os *Forms* são os elementos HTML por excelência para o envio/recepção de dados do cliente para o servidor.

• Estado:

- Como referido anteriormente, o protocolo HTTP é *stateless*, o que significa que não possui mecanismos de persistência de estado e por isso não permite a criação de sessões.
- Isto levou à implementação de mecanismos que sistemas web, exteriores ao protocolo HTTP, que permitem guardar dados de estado sobre múltiplas conexões entre um cliente e um servidor.
- Mecanismos:
 - Cookies
 - Ferramentas de alto nível, com recursos a base de dados, para gestão de sessões, utilizadores e registo

• Cookies:

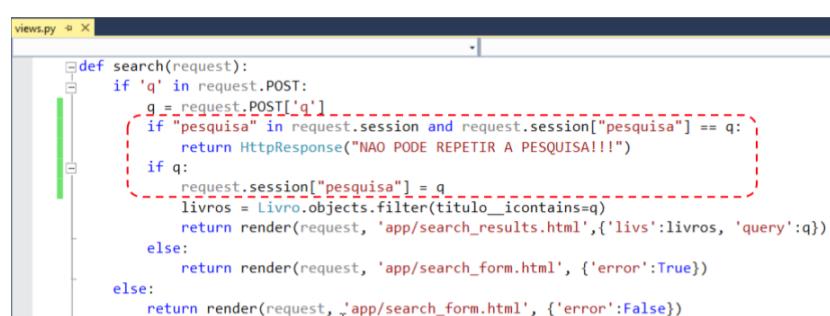
- Um *cookie* consiste num pequeno “pedaço” de informação que o servidor envia ao browser para este guardar, pelo menos, enquanto estes comunicam.
- É possível guardar dados diversos neste *cookie*, como o *username* do utilizador que fez o login, por exemplo.
- Embora o uso de *cookies* esteja generalizado pela grande maioria dos sites web, este padece de desvantagens:
 - A salvaguarda de *cookies* no browser é voluntária, o que não permite oferecer garantias do seu uso correto;
 - Não podem conter informações importantes, pois para além da falta de segurança a que estão sujeitos, o servidor pode ver-se inibido de aceder a informação crucial para a continuação da interação entre clientes e servidor.

• Sessões no Django:

- O Django oferece um mecanismo de alto nível para o estabelecimento de sessões, que permite guardar todo o tipo de informação de estado do lado do servidor.
 - Esta informação é guardada na base de dados
 - Para evitar este mecanismo, é necessário verificar que se encontra o seguinte no ficheiro “*settings.py*”

```
MIDDLEWARE_CLASSES = [
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
]
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'app',
)
```

- As sessões são geridas automaticamente pelo Django e este oferece um modo simples e “limpo” de gerir a informação de estado da sessão através do atributo “*request.session*” que consiste num dicionário.



```
def search(request):
    if 'q' in request.POST:
        q = request.POST['q']
        if "pesquisa" in request.session and request.session["pesquisa"] == q:
            return HttpResponse("NAO PODE REPETIR A PESQUISA!!!")
        if q:
            request.session["pesquisa"] = q
            livros = Livro.objects.filter(titulo__icontains=q)
            return render(request, 'app/search_results.html', {'livros':livros, 'query':q})
        else:
            return render(request, 'app/search_form.html', {'error':True})
    else:
        return render(request, 'app/search_form.html', {'error':False})
```

- O Django também oferece um mecanismo de alto nível para a autenticação de utilizadores
- Para ativar este mecanismo, é necessário verificar que se encontra o seguinte no ficheiro “settings.py”
 - Esta informação é guardada na base de dados

```

MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
)

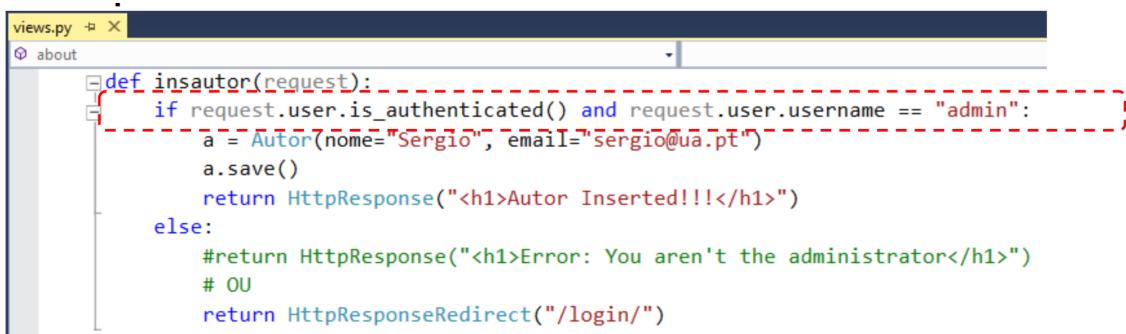
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'app',
)

```

- No caso de ainda não ter sido feito, deve-se executar o seguinte comando para iniciar a BD:

python manage.py syncdb

- A autenticação também é gerida automaticamente pelo Django e oferece acesso através do objeto “request.user”



```

views.py  ✘ X
about

def insautor(request):
    if request.user.is_authenticated() and request.user.username == "admin":
        a = Autor(nome="Sergio", email="sergio@ua.pt")
        a.save()
        return HttpResponseRedirect("<h1>Autor Inserted!!!</h1>")
    else:
        #return HttpResponseRedirect("<h1>Error: You aren't the administrator</h1>")
        # OU
        return HttpResponseRedirect("/login/")

```

- Documentação: <https://docs.djangoproject.com/en/1.8/topics/auth/default/>