

Tecnologias e Programação Web

José Mendes 107188

2023/2024

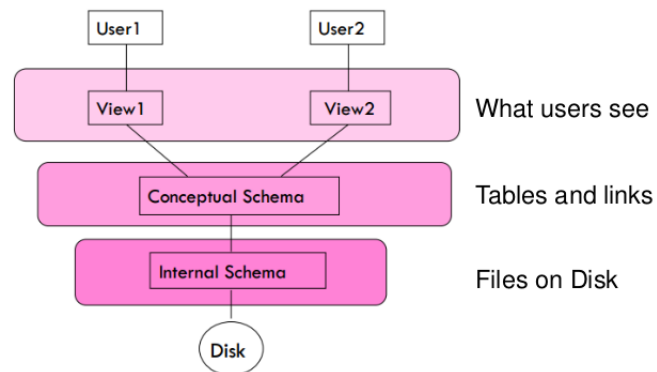


universidade
de aveiro

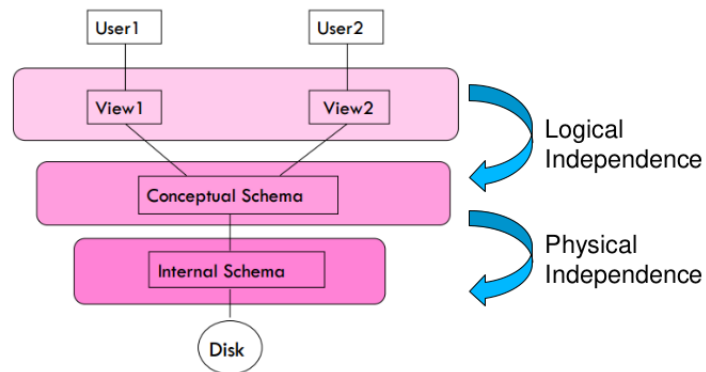
1 Arquiteturas de Aplicações Web

1.1 Independência dos Dados nas Bases de Dados

1.1.1 Arquitetura da Base de Dados e Views



1.1.2 Independência Lógica e Física



Cada nível é independente dos níveis a baixo.

1.1.3 Independência dos Dados

Independência Física: Capacidade de alterar o schema interno sem alterar o schema conceptual.

- Espaço de armazenamento pode mudar;
- O tipo de alguns dados pode mudar por razões de eficiência/otimização;

Independência Lógica: Capacidade de alterar o schema conceptual sem alterar as Views ou os programas de aplicação.

- Pode adicionar novos campos, novas tabelas sem alterar as Views;
- Pode alterar a estrutura das tabelas sem alterar as Views;

Nota: Manter a **View** (aquilo que o user vê) independente do **Modelo** (domain knowledge).

1.2 Arquiteturas N-Tier

1.2.1 Significado de "Tiers"

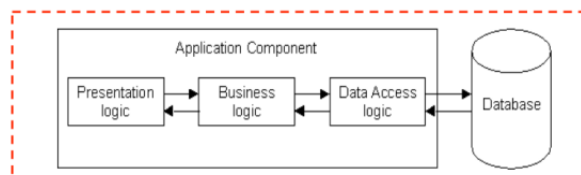
Arquiteturas N-Tier têm as mesmas camadas:

- **Presentation Tier:** Camada de apresentação;
- **Business/Logic Tier:** Camada de lógica/de negócio;
- **Data Tier:** Camada de dados;

Arquiteturas N-Tier tentam separar as camadas em diferentes tiers.

- **Camada:** Separação lógica;
- **Tier:** Separação física;

1.2.2 Arquitetura 1-Tier

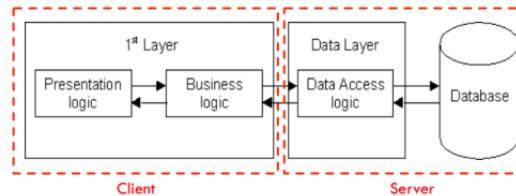


As 3 camadas estão no mesmo computador (máquina). Isto é, todo o código e processamento está numa única máquina.

As camadas de apresentação, lógica e de dados estão firmemente conectados.

- **Escalabilidade:** Um único processador implica que é difícil de aumentar o volume de processamento;
- **Portabilidade:** Mudar para outro computador pode implicar reescrever tudo;
- **Manutenção:** Mudar uma camada requer mudar outras camadas;

1.2.3 Arquitetura 2-Tier

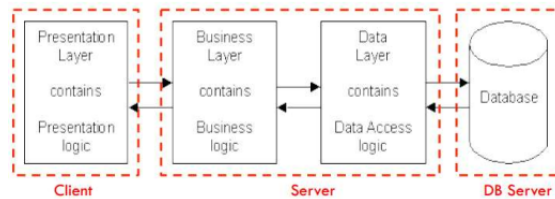


A base de dados corre no servidor, separado do cliente, sendo fácil de mudar para uma base de dados diferente.

As camadas de apresentação e lógica ainda estão firmemente conectados.

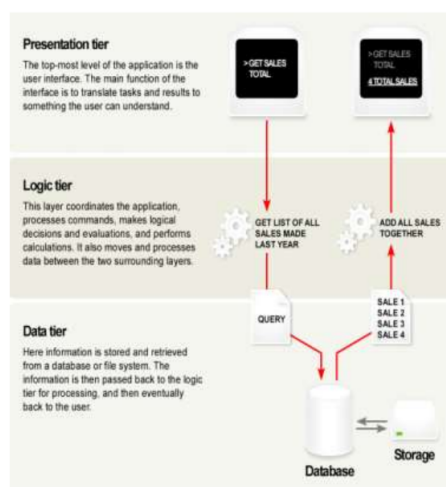
- Carga pesada no servidor;
- Potencial congestionamento da rede;
- Apresentação ainda está firmemente conectada à lógica de negócio;

1.2.4 Arquitetura 3-Tier



Cada camada pode, potencialmente, correr numa máquina diferente. As camadas de apresentação, lógica e de dados estão desconectadas.

Típica Arquitetura 3-Tier



Princípios de Arquitetura:

- Arquitetura **Cliente-Servidor**;
- Cada camada (Apresentação, Lógica, Dados) deve ser independente e não deve expor dependências relacionadas com a implementação;
- Camadas desconectadas não devem comunicar;
- Alterações numa plataforma afetam apenas a camada que corre que essa plataforma;

Camada de Apresentação:

- Fornece a interface do utilizador;
- Lida com a interação com o utilizador;
- Por vezes chamada de GUI, client view ou **front-end**;
- Não deve conter lógica de negócio ou código de acesso aos dados;

Camada de Lógica:

- O conjunto de regras para processar a informação;
- Pode acomodar vários utilizadores;
- Por vezes chamada de **middleware** ou back-end;
- Não deve conter apresentação ou código de acesso aos dados;

Camada de Dados:

- A camada de armazenamento físico, para persistir os dados;
- Gere o acesso à base de dados ou sistema de ficheiros;
- Por vezes chamada de **back-end**;
- Não deve conter apresentação ou código da lógica de negócio;

1.2.5 Arquitetura 3-Tier para Aplicações Web

Camada de Apresentação: Conteúdo estático ou dinâmico, renderizado pelo browser (**front-end**);

Camada de Lógica: Processamento de conteúdo dinâmico, geração de servidores de aplicação (e.g. Java EE, ASP.NET, Python Django Framework) (**middleware**);

Camada de Dados: Base de dados, composta por ambos, conjunto dos dados e o sistema de gestão de base de dados ou software RDBMS que gere e fornece acesso aos dados (**back-end**);

1.2.6 Arquitetura 3-Tier - Vantagens

Independência das camadas

- Mais fácil de manter;
- Componentes são reutilizáveis;
- Desenvolvimento mais rápido (divisão do trabalho, Web Designer faz a apresentação, Engenheiros de Software fazem a lógica e administradores de base de dados fazem o modelo de dados);

1.3 Padrões de Desenho

1.3.1 Padrões de Desenho e Decisões

- Construção e teste;
 - como construímos uma aplicação web?
 - que tecnologias devemos escolher?
- Reutilizar;
 - podemos usar componentes normais?
- Escalabilidade;
 - como vai a nossa aplicação web lidar com um elevado número de pedidos?
- Segurança;
 - como proteger contra um attack, vírus, acesso malicioso dos dados, denial of service?
- Diferentes views de dados;
 - tipos de users, contas individuais, proteção de dados

Nota: Precisamos de uma solução geral e reutilizável: **Padrões de Desenho**

1.3.2 O que é um Padrão de Desenho?

Uma solução geral e reutilizável para um problema recorrente no desenho de software. É um template para como resolver um problema que tenha sido usado em várias situações diferentes. **NÃO** é um design completo, o padrão precisa de ser adaptado à aplicação, não podemos simplesmente traduzir o padrão para código.

1.3.3 Origem do Padrão de Desenho

- Arquitetura conceptual por Christopher Alexander (1977/1979).
- Adaptado para programação OO por Kent Beck e Ward Cunningham (1987).
- Ganhou popularidade em CS com o livro "Design Patterns: Elements of Re-usable Object-Oriented Software" (1994), por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (Gang of Four).
- Agora bastante utilizado em Engenharia de Software.

1.4 O Padrão de Desenho MVC

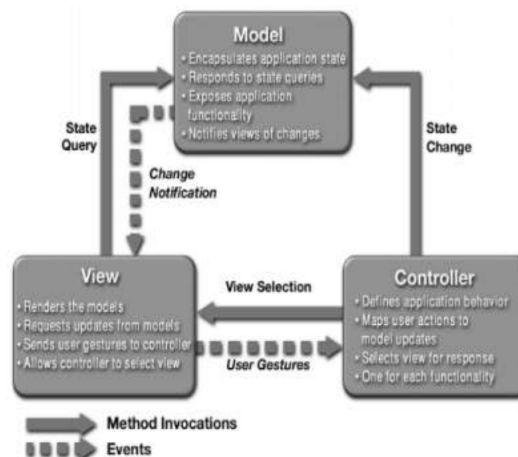
1.4.1 O Problema do Desenho

- Preciso mudar o look-and-feel sem mudar o core/logic;
- Preciso de apresentar os mesmos dados de diferentes formas (e.g. computadores bons, web, dispositivos móveis);
- Preciso de interagir/acessar os dados de diferentes formas (e.g. ecrã tátil nos dispositivos móveis, teclado no computador);
- Preciso de manter várias views para a mesma informação (e.g lista, thumbnails, detalhes, ...);

1.4.2 A Solução do Desenho

- Separar a funcionalidade chave da apresentação e lógica que usa esta funcionalidade;
- Permitir múltiplas views para partilhar o mesmo modelo de dados;
- Tornar o suporte de vários clientes mais fácil de implementar, testar e manter;

1.4.3 O Padrão Model-View-Controller



Padrão de Desenho para sistemas gráficos que **promove a separação entre o modelo e a view**. Com este padrão, a lógica necessária para manutenção dos dados (base de dados, ficheiro de texto), **é separada** de como os dados são apresentados (gráfico, numerico) e como os dados são manipulados (GUI, linha de comandos, touch).

Model

- Gere o comportamento e os dados do domínio da aplicação;
- Responde a pedidos por informação sobre o estado (normalmente da view), segue instruções para alterar o estado (normalmente do controller);

View

- Renderiza o modelo para uma forma apropriada para a interação, tipicamente uma interface do utilizador (várias views podem existir para o mesmo modelo com diferentes intenções);

Controller

- Recebe os inputs e inicia uma resposta realizando chamadas em objetos do modelo;
- Aceita input do utilizador e intrui o modelo e a view para realizar ações baseadas no input;

1.4.4 O Padrão MVC na prática

Model

- Contém conhecimento específico do domínio;
- Regista o estado da aplicação (e.g quais items estão no carrinho de compras);
- Normalmente conectado com a base de dados;
- Independente da view (um model pode ter várias views);

View

- Apresenta dados ao utilizador;
- Permite interação com o utilizador;
- Não faz processamento;

Controller

- Define como a interface do utilizador reage a inputs (eventos);
- Recebe mensagens da view (de onde os eventos vêm);
- Envia mensagens ao modelo (diz quais dados mostrar)

1.4.5 O MVC para Aplicações Web

Model

- Tabelas da base de dados (dados persistentes);
- Informações sobre a sessão (dados atuais do sistema de dados)
- Regras sobre transações;

View

- (X)HTML;
- CSS style sheets;
- Templates server-side;

Controller

- Scripts client-side;
- Processamento de pedidos HTTP;
- Lógica de negócio/preprocessamento;

1.4.6 MVC - Vantagens

Clareza de Design

- métodos do modelo dão uma API para os dados e o estado;
- o design da view e do controller são fáceis;

Modularidade eficiente

- qualquer componente pode ser facilmente substituído;

Várias views

- Várias views podem ser criadas conforme apropriado;
- Cada uma usa a mesma API para o modelo;

Mais fácil de contruir e manter

- Views simples (baseado em texto) durante o desenvolvimento (construção);
- Mais views e controllers podem ser adicionados mais tarde;
- Fácil de contruir interfaces estáveis;

Distribuível

- Ajuste natural para ambientes distribuídos;

1.4.7 3-Tier Architecture vs Padrão MVC

- Comunicação
 - **3-Tier:** A camada de apresentação nunca comunica diretamente com a camada de dados, apenas através da camada de lógica (topologia linear);
 - **MVC:** Todas as camadas comunicam diretamente (topologia triangular);
- Usabilidade
 - **3-Tier:** É usada em aplicações web onde os tiers cliente, middleware e os dados corram em plataformas fisicamente separadas;
 - **MVC:** Historicamente usada em aplicações que correm numa única estação de trabalho gráfica;
 - * No contexto de aplicações web, contudo, os componentes lógicos podem ser desacoplados para cumprir com uma verdadeira arquitetura 3-Tier;

2 Introdução à Plataforma Django

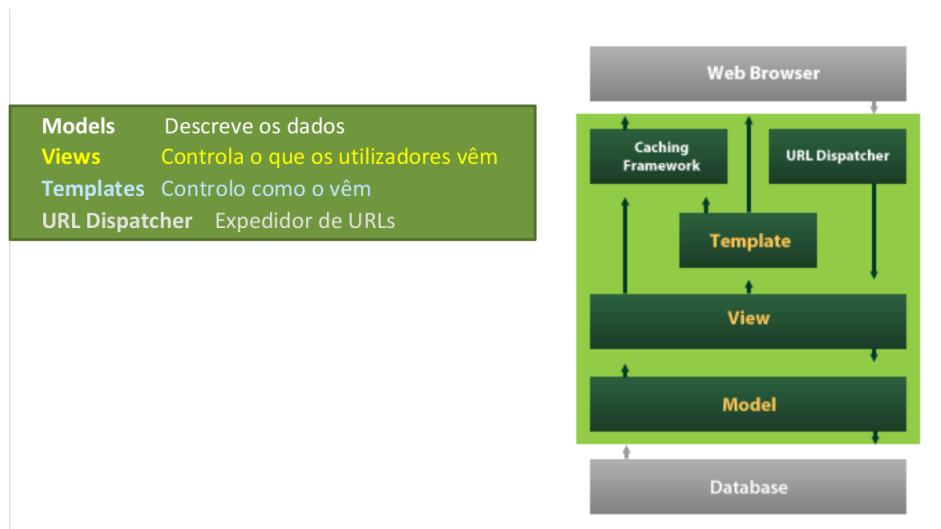
2.1 Introdução

Django é uma plataforma gratuita e open-sourc, escrita em Python, para o desenvolvimento de aplicações web. Tem o nome do famoso guitarrista de jazz Django Reinhardt. É mantida pela Django Software Foundation (DSF), uma organização independente. Fomenta o desenvolvimento rápido, limpo e pragmático. Criada em 2003, lançada open-source em 2005.

2.2 Características

- Segue, parcialmente, o padrão MVC;
- Possui um ORM (Object-Relational Mapper) para processar dados;
- Focada na automatização, aderindo ao princípio DRY (Don't Repeat Yourself);
- Usa um sistema de templates;
- Sistema de personalização Admin, para facilitar o CRUD;
- Desenho elegante de routing de URLs;
- Possui um light web server embotido (para testes);
- Possibilita a utilização de moddleware personalizado;
- Possui facilidades para: autenticação, internacionalização, caching;

2.3 Arquitetura



2.4 Estrutura de um Projeto Django

```
webproj/ ----- Pasta para o projeto. Pode ter qualquer nome.
manage.py -- Utilitário em commando de linha para interagir com o projeto.
webproj/ --- Pacote do projeto. Nome usado para imports.
__init__.py --- Ficheiro que define esta pasta como um pacote, em Python.
settings.py --- Configurações do projeto Django.
urls.py ----- Mapping/routing das URLs para este projeto.
wsgi.py ----- Um ponto de entrada para webserver compatíveis com WSGI.
app/ ----- Aplicação web individual, podendo coexistir várias.
templates/ ---- Ficheiros HTML, invocados pelas views.
static/ ----- CSS, JS, imagens, etc. - configurável em "settings.py"
__init__.py -- Ficheiro que define esta pasta como um pacote, em Python.
views.py ----- Recebe os pedidos dos clientes e devolve as respostas.
models.py ----- Modelos dos dados.
admin.py ----- Criação automática de interface para o modelo de dados.
forms.py ----- Permite a receção de dados enviados pelos clients.
```

2.5 Settings

Possui um ficheiro de configuração, **settings.py**, do projeto Django, sobrepõe as configurações padrão (ficheiro `python/lib/sitepackages/django/conf/global_settings.py`).

Possui alguns atributos como: **DEBUG** (True/False), **DATABASES ENGINES** (sqlite3, mysql, postgresql, oracle), **ROOT_URLCONF** (nome do ficheiro de routing de URLs), **MEDIA_ROOT** (para ficheiros enviados pelo utilizador), **MEDIA_URL** (para ficheiros multimédia), **STATIC_ROOT** (pasta para ficheiros estáticos, CSS, JS), **STATIC_URL** (pasta de ficheiros estáticos), **TEMPLATE_DIRS** (pasta de templates)

2.6 Criação de um Projeto Django

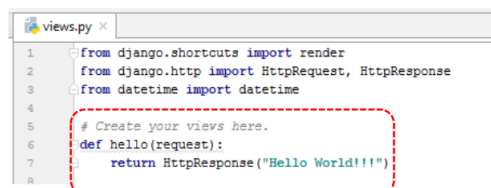
Nesta cadeia vamos usar o PyCharm, pelo que para criar um projeto Django basta seleccionar a opção **Django** e chamar a pasta dos templates de **app/templates** e a pasta da aplicação de **app**.

Para correr basta clicar em Run e aceder ao link **http://127.0.0.1:8000**.

Ou podemos usar o comando **python manage.py runserver**.

2.6.1 Views

No ficheiro "app/views.py" podemos inserir views:



```
1 from django.shortcuts import render
2 from django.http import HttpRequest, HttpResponse
3 from datetime import datetime
4
5 # Create your views here.
6 def hello(request):
7     return HttpResponse("Hello World!!!")
```

Para criar uma nova, basta criar uma nova view function.

2.6.2 Configuração da URL

No ficheiro "nome_do_projeto/urls.py" podemos configurar as URLs:



```
14 2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
15
16 from django.contrib import admin
17 from django.urls import path
18
19 from app import views
20
21 urlpatterns = [
22     path('hello/', views.hello, name='hello'),
23 ]
```

Para criar uma nova, basta inserir mais uma route para a view.

2.6.3 Templates

Podemos usar variáveis usando `{{ var_name }}` e template tags (como if) usando `{% if ... %}`.

2.6.4 Ficheiros estáticos

São ficheiros que se pretende simplesmente referenciar e servir ao cliente, sem qualquer processamento prévio.

O seu acesso é público, pois o cliente apenas necessita do URL para os mesmos.

Exemplos:

- Imagens (jpg, png, gif, ...);
- Style Sheets (CSS);
- Scripts (JavaScript);
- ...

Localização

Os ficheiros denominados por static files residem em pastas pré-determinadas, dentro ou fora da “app” (normalmente uma pasta denominada de “static”).

Configuração

No ficheiro “settings.py” podemos configurar os ficheiros estáticos:

1. o módulo ‘django.contrib.staticfiles’ deve aparecer nas aplicações instaladas “INSTALLED_APPS”
2. o atributo STATIC_URL deve ser definido (ex: ‘/static/’)
3. a pasta dos static files deve estar definida no atributo STATIC_ROOT (ex: os.path.join(BASE_DIR, ‘app/static’))

Uso

Para usar basta usar a tag {% load static %} e depois referenciar o ficheiro.

2.7 Modelos - Camada de Base de Dados Django

2.7.1 Modelo

- MTV - Model-Template-View;
- Modelo
 - Consiste na camada de acesso aos dados - “Data Access Layer”;
 - Esta camada permite definir, em relação aos dados: o **Acesso**, a **Validação**, o **Comportamento** e as **Relações** entre os dados;

2.7.2 Configuração da Base de Dados

No ficheiro "settings.py" podemos configurar a base de dados, basta procurar pela variável **DATABASES** e definir os parâmetros **ENGINE**, **NAME**, **USER**, **PASSWORD**, **HOST**, **PORT**.

Por defeito, o Django usa o SQLite3.

2.7.3 Criação de um modelo

A criação de modelos vai acontecer no ficheiro "app/models.py". Para criar um modelo basta criar uma classe que herde de **models.Model**. Os atributos da classe são os campos da tabela.

Para cada atributo das classes é instanciado um objeto tipo "Field" e/ou subtipo, como: **CharField**, **DataField**, etc.

Alguns atributos, representam a criação de relações entre as classes, tendo como efeito a criação de colunas com chaves estrangeiras (**1:1**, **1:M**, **M:1**) ou de tabelas de associação (**M:N**)

Relações entre classes:

- **1:M e M:1**
 - Conseguída com o atributo **models.ForeignKey**;
 - Exemplo: Publisher (1) : (M) Book ou Book (M) : (1) Publisher O atributo é colocado na classe que representa o lado "M" da relação, neste caso a classe Book;
- **1:1 (único)**
 - Conseguída com o atributo **models.OneToOneField**;
 - Exemplo: Book (1) : (1) Place O atributo "deve" ser colocado na classe que mais necessita da outra, neste caso a classe Book;
- **M:N**
 - Conseguída com o atributo **models.ManyToManyField**;
 - Exemplo: Book (M) : (N) Author O atributo "deve" ser colocado na classe que mais "necessita" da outra, como na relação de 1:1

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) da tabela.

Com vista à ativação do modelo, a aplicação web, "app", deve ser incluída na variável **INSTALLED_APPS** do ficheiro "settings.py". Caso a aplicação já esteja a ser incluída, de forma indireta, não é necessário.

2.7.4 Comandos na criação de um modelo

python manage.py check - valida o modelo (sintaxe e lógica)

python manage.py makemigrations - produz código de migração (na pasta app/migrations devemos apagar tudo menos o ficheiro __init__.py)

python manage.py sqlmigrate app 0001 - mostra o SQL gerado pela migração

python manage.py migrate - produz as tabelas na BD

2.7.5 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 tal, é necessário no ficheiro "urls.py" da aplicação adicionar:

- o import "**from django.contrib import admin**"
- a linha "**path('admin/', admin.site.urls)**"

Também é necessário registar as classes que se pretende gerir:

- o mesmo import que em cima
- importar as classes que se pretende gerir de "**app.models**"
- adicioná-los: "**admin.site.register(Class)**"

Por fim é necessário criar a conta de administração, com o comando: **python manage.py createsuperuser**

Programando:

- Inserir um objeto:

```
a = Author(name="Jose", email="mendes.j@ua.pt")
a.save()
```

- Modificar um objeto:

```
a.email = "ze.mendes@ua.pt"
a.save()
```

- Selecionar todos os objetos:

```
Author.objects.all()
```

-
- Filtrar objetos (por nome):

```
Autor.objects.filter(name="Jose")
```

- Filtrar por nome e por email:

```
Autor.objects.filter(name="Jose", email="mendes.j@ua.pt")
```

- Filtrar por nome parecido:

```
Autor.objects.filter(name__contains="Jose")
```

- Aceder a um único objeto:

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

- Ordenação:

```
Publisher.objects.order_by("city", "country")
```

- Filtragem e Ordenação:

```
Publisher.objects.filter(country="Portugal").order_by("-city")
```

- Selecionar os primeiros resultados:

```
Publisher.objects.order_by("city", "country")[0]  
Publisher.objects.order_by("city", "country")[0:4]
```

– **Nota:** Não são permitidos índices negativos.

- Remover um objeto:

```
a = Author.objects.get(email="mendes.j@ua.pt").delete()
```

2.8 Mandar e Receber Dados - Forms

2.8.1 Receber Dados

O objeto **request** do tipo **HttpRequest** permite aceder a um vasto conjunto de dados, recebidos pelo servidor web.

Estes dados podem ser diretamente acessados através de alguns métodos e atributos como: **request.path**, **request.get_host()**, **request.is_secure()**

Ou podem ser acessados pelo dicionário **request.META**, que contém toda a informação presente no cabeçalho do protocolo HTTP.

Exemplo mostrando todos os dados presentes no cabeçalho HTTP:



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

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 HttpResponse('<table>%s</table>' % '\n'.join(html))
```

2.8.2 Forms

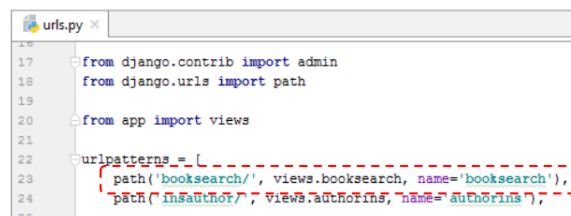
Por excelência, os forms são os elementos HTML para enviar e receber dados do cliente para o servidor.

Do lado do cliente (browser), o Form pode usar os métodos **GET** ou **POST** para enviar os dados que possui.

Do lado do servidor, a view chamada pode usar os dicionários "**request.GET**" e "**request.POST**" para aceder aos dados recebidos.

Exemplo: Form para pesquisar livros por título

Definir o URL:



```
urls.py x
16
17 from django.contrib import admin
18 from django.urls import path
19
20 from app import views
21
22 urlpatterns = [
23     path('booksearch/', views.booksearch, name='booksearch'),
24     path('insauthor/', views.authorins, name='authorins'),
25 ]
```

Definir a view:

```
views.py
5 from app.models import Author, Publisher, Book
6
7
8 def booksearch(request):
9     if 'query' in request.POST:
10         query = request.POST['query']
11         if query:
12             books = Book.objects.filter(title__icontains=query)
13             return render(request, 'booklist.html', {'books':books, 'query':query})
14         else:
15             return render(request, 'booksearch.html', {'error':True})
16     else:
17         return render(request, 'booksearch.html', {'error':False})
```

Definir o template para pesquisa:

```
booksearch.html
1 {% extends "layout.html" %}
2
3 {% block content %}
4
5 {% if error %}
6     <p style="...">ERROR: Insert a query term.</p>
7 {% endif %}
8
9 <form action="." method="post">
10     {% csrf_token %}
11     <input type="text" name="query">
12     <input type="submit" value="Search">
13 </form>
14
15 {% endblock %}
```

Definir o template para mostrar os resultados:

```
booklist.html
1 {% extends "layout.html" %}
2 {% block content %}
3 <p>Search by: <strong>{{ query }}</strong></p>
4 {% if books %}
5 <p>Found {{ books|length }} book{{ books|pluralize }}.</p>
6 <ul>
7     {% for bok in books %}
8         <li>{{ bok.title }}</li>
9         <ul>
10             <li>{{ bok.publisher }}</li>
11             <li>{{ bok.date }}</li>
12             <ul>
13                 {% for aut in bok.authors.all %}
14                     <li>{{ aut }}</li>
15                 {% endfor %}
16             </ul>
17         </ul>
18     {% endfor %}
19 </ul>
20 {% else %}
21 <p>Not found any result.</p>
22 {% endif %}
23 {% endblock %}
```

2.8.3 Classes Form e Django Forms

A classe **Form** descreve um formulário e determina como ele se comporta e como é apresentado no browser.

Os campos da classe **Form** mapeiam para o formulário HTML como elementos `<input>`.

- Eles próprios são classes, que gerem os dados de um formulário e fazem a validação quando o formulário é submetido;
- São representados no browser como um "widget" HTML. Cada campo tem um default widget apropriado, mas estes podem ser overridden, se necessário;

2.8.4 Instância de um Form

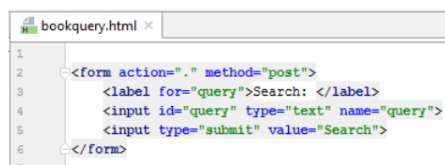
Numa instância de uma classe **Form**, podemos optar por deixar vazio ou pre-popular, por exemplo com:

- dados de uma instância de um modelo salvo (como é o caso do Django Admin para editar);
- dados que tenhamos coletado de outras fontes;
- dados recebidos de uma submissão anterior de um formulário HTML;

O último caso é muito útil, pois permite os utilizadores reenviar informação sem terem que a reescrever.

2.8.5 Contruir um Form

Para contruir um formulário, normalmente escrevemos o código em HTML assim:

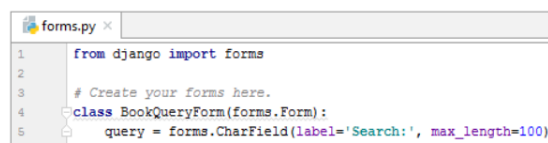


```
1 <form action="." method="post">
2   <label for="query">Search: </label>
3   <input id="query" type="text" name="query">
4   <input type="submit" value="Search">
5 </form>
```

De facto, um formulário é bem mais complexo, incluindo vários campos, tipos de campos, restrições e regras de validação, pelo que seria bom ser mais fácil

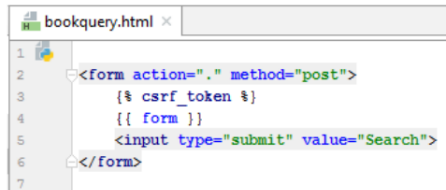
2.8.6 Criando um Django Form

Para criar uma classe **Form** no Django, basta criar uma classe que herde de **forms.Form**, no ficheiro "app/forms.py".



```
1 from django import forms
2
3 # Create your forms here.
4 class BookQueryForm(forms.Form):
5     query = forms.CharField(label='Search:', max_length=100)
```

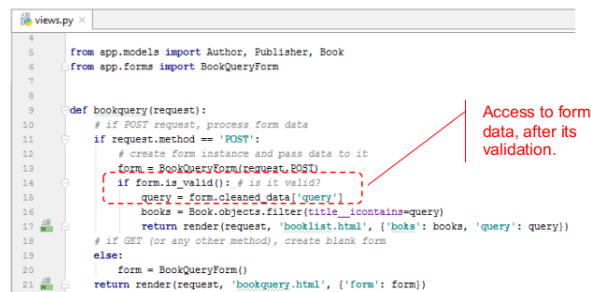
Depois criamos o template onde a class **Form** vai ser representada:



```
1
2 <form action="." method="post">
3     {% csrf_token %}
4     {{ form }}
5     <input type="submit" value="Search">
6 </form>
7
```

Quando renderizado, o formulário vai substituir o template tag `{{ form }}` com o label e o input definidos na classe **Form**.

A view será do tipo:



```
4
5 from app.models import Author, Publisher, Book
6 from app.forms import BookQueryForm
7
8
9 def bookquery(request):
10     # if POST request, process form data
11     if request.method == 'POST':
12         # create form instance and pass data to it
13         form = BookQueryForm(request.POST)
14         if form.is_valid(): # is it valid?
15             query = form.cleaned_data['query']
16             books = Book.objects.filter(title__icontains=query)
17             return render(request, 'booklist.html', {'books': books, 'query': query})
18     # if GET (or any other method), create blank form
19     else:
20         form = BookQueryForm()
21         return render(request, 'bookquery.html', {'form': form})
```

Os Campos

Os campos dos dados:

- Os dados submetidos com o formulário, usando **Form Fields**, podem ser validados usando a função **is_valid()**;
- Depois da validação, os dados podem ser acessados no dicionário **form.cleaned_data**;
- Os dados no dicionário já estão convertidos para tipos Python para uso imediato;

Exemplo de Data Fields e a sua representação HTML:

- **BooleanField** - Checkbox Input;
- **CharField** - Text Input;
- **IntegerField** ou **FloatField** - Number Input;
- **DateField** e **TimeField** - Text Input;
- **ChoiceField** - Select;
- **MultipleChoiceField** - Select Múltiplo;
- **FileField** - File Input;

Para já não é o mais aesthetically pleasing, mas corre direito e faz validação automática.

É possível renderizar os Form Fields individualmente:



```
1  {% extends "layout.html" %}
2
3  {% block content %}
4
5      <h2>{{ title }}</h2>
6      <div class="row">
7          <div class="col-md-8">
8              <section id="insbookForm">
9                  <form action="." method="post" class="form-horizontal">
10                     {{ csrf_token }}
11                     <h4>Insert query to search book titles.</h4>
12                     <hr />
13                     <div class="form-group">
14                         {{ form.query.label_tag }}
15                         <div class="col-md-10">
16                             {{ form.query }}
17                         </div>
18                     </div>
19                     <div class="form-group">
20                         <div class="col-md-offset-2 col-md-10">
21                             <input type="submit" value="Search" class="btn btn-default" />
22                         </div>
23                     </div>
24                 </form>
25             </section>
26          </div>
27      </div>
28  {% endblock %}
```

2.9 Autenticação Django

O Django vem com um sistema de autenticação de utilizadores. Este sistema lida com contas de utilizadores, grupos, permissões e sessões de utilizadores.

Trata de ambos, a autenticação e a autorização.

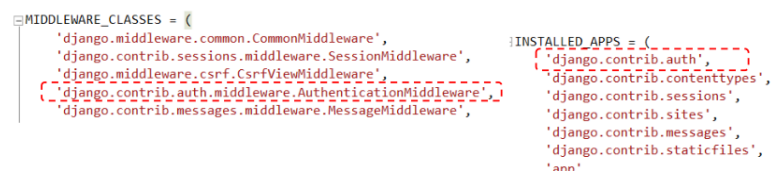
- **Autenticação** - Verificar que o utilizador é quem diz ser;
- **Autorização** - Determina o que o utilizador autenticado tem permissão para fazer;

Implementa:

- Utilizadores, grupos e permissões;
- Ferramentas de Forms e view para "loggar" utilizadores ou restringir conteúdo;
- Hashing de passwords;

2.9.1 Autenticação

Para usar o sistema de autenticação, temos de ver se os seguintes módulos estão incluídos:



```
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',
)
```

Além disso, a base de dados deve estar iniciada, senão: **python manage.py migrate**

2.9.2 Criar Login e Logout

No ficheiro "app/urls.py" devemos adicionar as seguintes linhas:

```
16
17 from django.contrib import admin
18 from django.contrib.auth import views as auth_views
19 from django.urls import path, include
20
21 from app import views
22
23 urlpatterns = [
24     path('login/', auth_views.LoginView.as_view(template_name='login.html'), name='login'),
25     path('logout', auth_views.LogoutView.as_view(next_page='/', name='logout'),
26
```

Usar o ficheiro "login.html" par criar o template do login, e fazer as seguintes alterações:

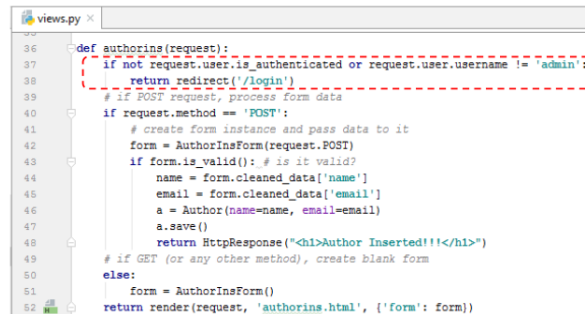
```
9
10 <form action="." method="post" class="form-horizontal">
11     {% csrf_token %}
12     <h4>Use a local account to log in.</h4>
13     <hr />
14     <div class="form-group">
15         {{ form.username.label_tag }}
16         <div class="col-md-10">
17             {{ form.username }}
18         </div>
19     </div>
20     <div class="form-group">
21         {{ form.password.label_tag }}
22         <div class="col-md-10">
23             {{ form.password }}
24     </div>
25 </div>
```

Usar o ficheiro "loginpartial.html" para criar a área se vai mostrar o status do login, para tal modificamos o ficheiro "layout.html":

```
27
28 <div class="collapse navbar-collapse" id="navbarDefault">
29     <ul class="navbar-nav mr-auto">
30         <li class="nav-item active">
31             <a class="nav-link" href="{% url 'home' %}">Home</a>
32         </li>
33         <li class="nav-item">
34             <a class="nav-link" href="{% url 'about' %}">About</a>
35         </li>
36         <li class="nav-item">
37             <a class="nav-link" href="{% url 'contact' %}">Contact</a>
38         </li>
39     </ul>
40     {% include 'loginpartial.html' %}
41 </div>
</nav>
```

2.9.3 Autorização

A autorização pode ser gerida automaticamente pelo Django, através do objeto "**request.user**", é possível verificar se um dado utilizador está autenticado e se tem permissão para fazer operações.



```
36 def authorins(request):
37     if not request.user.is_authenticated or request.user.username != 'admin':
38         return redirect('/login')
39     # if POST request, process form data
40     if request.method == 'POST':
41         # create form instance and pass data to it
42         form = AuthorInsForm(request.POST)
43         if form.is_valid(): # is it valid?
44             name = form.cleaned_data['name']
45             email = form.cleaned_data['email']
46             a = Author(name=name, email=email)
47             a.save()
48             return HttpResponse("<h1>Author Inserted!!!</h1>")
49     # if GET (or any other method), create blank form
50     else:
51         form = AuthorInsForm()
52     return render(request, 'authorins.html', {'form': form})
```

2.10 Sessões Django

2.10.1 Estado

O protocolo HTTP é um protocolo sem estado (stateless), o que significa que não tem nenhum mecanismo para guardar o estado da conexão e, desta forma, não permite a criação de sessões.

Para tal, alguns mecanismos externos foram desenvolvidos em clientes web e servidores, que permitem guardar o dados do estado em várias conexões HTTP, produzindo sessões artificiais.

Mecanismos como:

- Cookies;
- Ferramentas de alto nível, usando BDs para gerir utilizadores, autenticações e sessões;

2.10.2 Cookies

Uma cookie é uma peça de informação enviada pelo servidor web para o cliente, o browser, para ser guardada enquanto estão numa conexão.

É possível guardar algum tipo de informação nesta cookie, como o nome do utilizador por exemplo.

Este mecanismo de cookies é bastante usado por quase todos os web sites, mas possui algumas disvantagens:

- Salvar cookies no browser não é obrigatório, o que não permite oferecer garantia de um bom serviço;
- Cookies não podem ser usadas para guardar informação importante - não são seguras;
- Os servidores podem ser inibidos, em algum momento, de acessar informação crucial para continuar a interação com o cliente.

2.10.3 Sessões Django

O Django oferece um mecanismo de alto nível para o estabelecimento de sessões, o que permite salvar todo o tipo de informações no próprio servidor. Esta informação é armazenada na base de dados.

Para usar este mecanismo, verifica se as seguintes linhas estão 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',
)
```

O Django gere automaticamente as sessões com os clientes de uma maneira simples e transparente. Utiliza o objeto **"request.session"** que é um dicionário.

```
views.py
def bookquery(request):
    # if POST request, process form data
    if request.method == 'POST':
        # create form instance and pass data to it
        form = BookQueryForm(request.POST)
        if form.is_valid(): # is it valid?
            query = form.cleaned_data['query']
            if 'searched' in request.session and request.session['searched'] == query:
                return HttpResponse('Query already made!!!')
            request.session['searched'] = query
            books = Book.objects.filter(title__icontains=query)
            return render(request, 'booklist.html', {'books': books, 'query': query})
    # if GET (or any other method), create blank form
    else:
        form = BookQueryForm()
        return render(request, 'bookquery.html', {'form': form})
```