Projeto - *Talking Formula 1*

Bases de Dados e Análise de Informação

Diogo Gonçalves[1], **Maria Miguel Cruz**[2]
Departamento de Física, UC

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

ID: 2018279462[1], 2018279834[2]
Janeiro 2022, Coimbra

Conteúdo

1 Auto-avaliação

1.1 a. Sucessos e Faltas

1.1.1 Sucessos

Fizemos um trabalho com uma imagem gráfica apelativa e, no geral, coesa. Mostramos que sabemos dispor os dados de diversas formas e todas elas bem implementadas. O Diagrama ER é sólido e robusto, permitindo explorar associações entre os diferentes dados de forma eficiente.

1.1.2 Falhas

Poderíamos ter incluído uma search bar e evoluir um pouco mais o `.html` e o `.css.` em alguns campos.

1.2 b. auto avaliação do grupo

De uma maneira geral, consideramos que merecemos 90%. Como dissemos anteriormente, ficou apenas a faltar implementar a `search bar`, mas no geral temos um trabalho suficientemente complexo e com elevada qualidade.

1.3 c. Lista do que cada um contribuiu

:
Maria:

- Conceptualização do projeto
- Diagrama ER
- Idealização do front-end
- Desenvolvimento views.py (Details Functions, maioritariamente)
- Desenvolvimento do .html e .css

Diogo:

- Conceptualização do projeto
- Diagrama ER
- Migração do servidor para AWS
- Desenvolvimento views.py e funções auxiliares
- Desenvolvimento do .html e .css

1.4 Auto avaliação de cada elemento

Maria: 85%
Diogo: 95%
Em média, 90%

1.5 Horas de esforço

- Maria: 35 horas
- Diogo: 40 horas

2 Motivação

A principal motivação foi trabalhar com um enredo de dados que fosse interessante. Nesse sentido surgiu a ideia de trabalhar com os dados de um campeonato de Fórmula 1 moderno. Para este tema existem uma série de entidades onde temos liberdade de explorar os seus dados como a forma dos apresentar, trazendo nesse sentido uma complexidade adicional que ambicionámos no nosso projeto.

3 Diagramas

O esquema ERD da nossa base de dados é bastante direto. Importante mencionar que o `Construtor` é comparável a um clube (por exemplo de futebol), a `Equipa` é única a cada época e pertence ao construtor e alberga os `pilotos` (que podem ser comparáveis aos jogadores de futebol). O `Resultado` é indicativo à performance que o piloto teve numa dada `Corrida` (os chamados *Grand Prix*).
Na figura 1 podemos ver o *Entity Relation (ER) Diagram*:

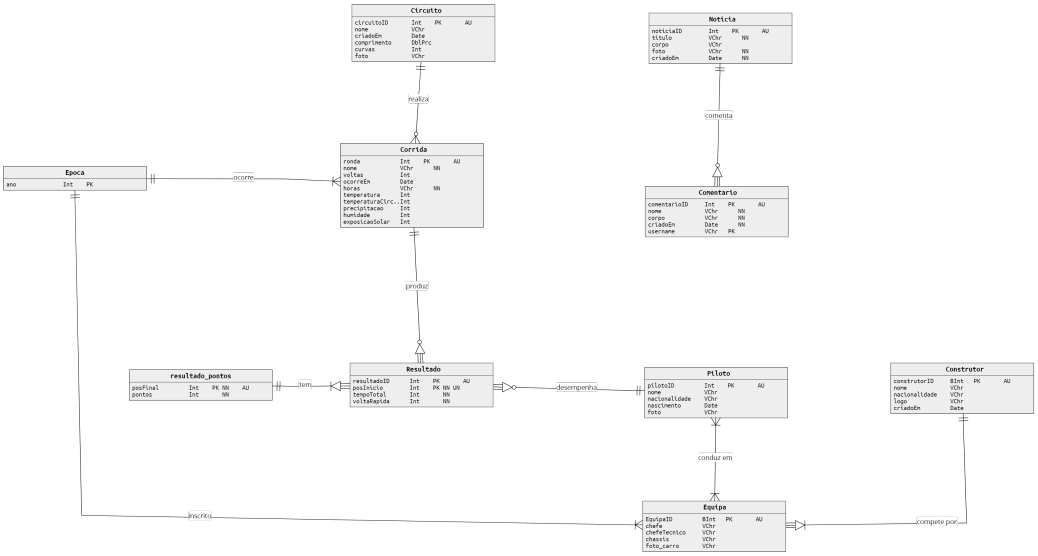

Figura 1: Este diagrama foi desenhado utilizando o http://onda.dei.uc.pt/v4/

Na figura 2 podemos ver o diagrama físico:

Figura 2: Este diagrama foi desenhado utilizando o http://onda.dei.uc.pt/v4/

4 Django

Iremos agora abordar o trabalho feito em Django.

4.1 settings.py

De forma a otimizar o fluxo do trabalho colocámos a base de dados na cloud (servidores gratuitos AWS). Segue em baixo o código que adicionamos no `settings.py`

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'talkingformuladb',
        'USER': 'postgres',
        'PASSWORD': 'adminadmin',
        'HOST': 'talkingformuladb.cjrbweuzqq2d.eu-west-3.rds.
           amazonaws.com',
        'PORT': '5432',
    }
}
```

4.2 models.py

Para este ficheiro, fizemos uso do package `pymysql` que permitiu criar o `models.py` à semelhança do gerado na aula. Importante denotar que fizemos uso da class interna `Meta` que permite aplicar funções aos objectos associados ao modelo.

Segue em baixo o uso da subclasse Meta para ordenar as épocas do ano maior para o mais pequeno:

```
class Epoca(models.Model):
    ano = models.IntegerField(primary_key=True)

    class Meta:
        managed = False
        db_table = 'epoca'
        ordering = ['-ano']

    def __str__(self):
        return self.ano
```

4.3 views.py e urls.py

Para o `views.py` iremos abordar singularmente todo o tipo de páginas geradas:

4.3.1 def index

O index define o conteúdo da nossa front page. Na figura 7 está um print do front-end que resulta do envio de todos os dados, sendo que iremos primeiramente abordar aqui o back-end e só mais à frente o front-end:

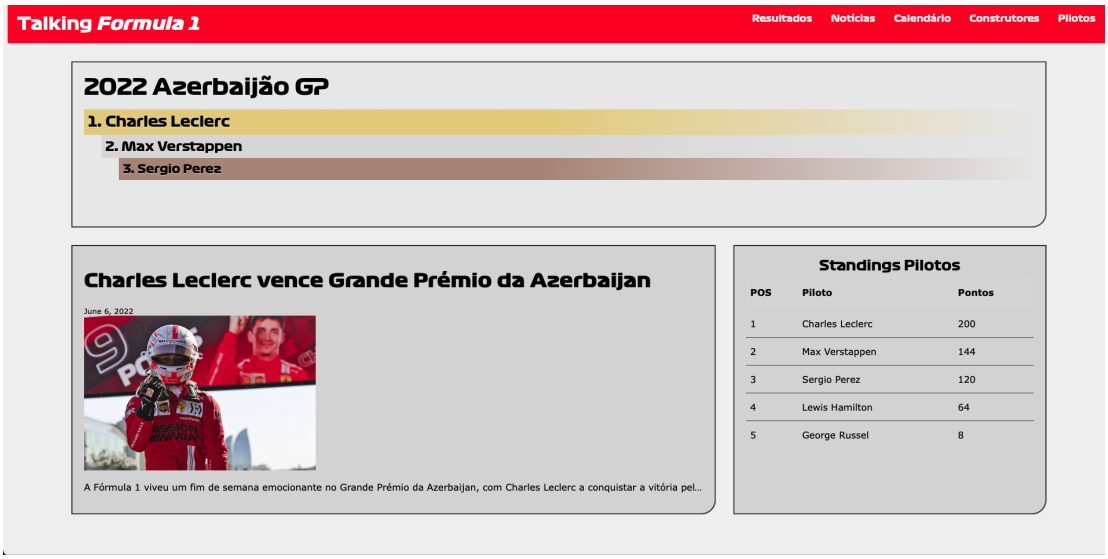

Figura 3: Print da primeira página, identificada como index

- Bloco topo - contém o nome da última corrida, o ano e os resultados dos três melhores qualificados, ou seja, o pódio. Os dados são obtidos através das variáveis `ultima_corrida` e `ultimo_resultados`;
- Bloco lado esquerdo - recebe o objeto relativo à última notícia, dado por `ultima_noticia`;
- Bloco lado direito - recebe a lista que resulta da função `calc_resultadosPilotos()`, a qual iremos abordar agora.

`calc_resultadosPilotos()`
Para esta função o objetivo é de obter o total de pontos (que são obtidos em função da `pos_final` e cuja relação é dada pela tabela `'resultados_pontos'`) para cada um dos pilotos e para a última época.

Assim, fizemos uma *query* dos dados utilizando as ferramentas do Django ORM:

```
1
2 from django.db.models import Sum
3 from django.db.models import Q
4
```

```
5  def calc_resultadosPilotos(epoca_ano=Epoca.objects.first().ano):
6
7      standings = np.empty((0, 2), dtype=object)
8      only_standings = np.empty(0, dtype=int)
9
10
11     for pilotoEquipa in (PilotoEquipa.objects.filter(Q(
           equipa_equipaid__epoca_ano__ano=epoca_ano))):
12
13         # Calculo da soma posfinal.pontos para o piloto_pilotoid
14         total_pontos = Resultados.objects.filter(piloto_pilotoid=
               pilotoEquipa.piloto_pilotoid).aggregate(Sum('
               posfinal__pontos'))['posfinal__pontos__sum']
15
16         # Adicao do piloto_pilotoid e total_pontos a array
17         standings = np.vstack([standings, [pilotoEquipa,
               total_pontos]])
18         only_standings = np.append(only_standings, total_pontos)
19
20     sorted_indices = np.argsort(only_standings)[::-1]
21     standings = standings[sorted_indices]
22
23
24     return standings
```

Analisando detalhadamente o código:

- `standings = np.empty((0, 2), dtype=object)` - matriz para guardar os dados;
- `only_standings = np.empty(0, dtype=int)` - matrix auxiliar para ordenar a matrix `standings`;
- `for pilotoEquipa in (PilotoEquipa.objects.filter(Q(equipa_equipaid__epoca_ano__ano = epoca_ano))):` - loop que irá iterar cada row da subEntidade `PilotoEquipa`, mas apenas para a época desejada, definida pela variável `epoca_ano`. Utilizamos o método `Q()`, pois este permite fazer consultas complexas;
- `total_pontos = Resultados.objects.filter(piloto_pilotoid = pilotoEquipa.piloto_pilotoid)['posfinal__pontos__sum']` - calcula o total de pontos para cada piloto. Para tal é feito primeiro um filtro para obter todos os resultados de um piloto de uma dada equipa e de seguida é feita uma agregação utilizando a função `Sum` para calcular a soma dos valores dos campos `posfinal__pontos` para todos os objetos retornados pela consulta. Importante mencionar que `posfinal__pontos` acede aos pontos relativos à dada posfinal, definido na entidade `resultado_pontos`. Por fim,

como é gerado um dicionário na agregação, o valor da soma de todos os pontos para um dado piloto está indexado por `['posfinal__pontos__sum']`.

4.3.2 `def pilotos` — `def circuitos` — `def construtores` — `def corridas` — `def noticias`

Para todas estas funções o conceito foi de dar *display* do conteúdo (ou seja, os objetos) de cada um dos respetivas modelos associados.

Tomando o exemplo de `def pilotos`, a função carrega o template do .html e os pilotos (`pilotos`) a serem mostrados. Por fim a função terá de dar `return` destes dois elementos.

```
def pilotos(request):
    template = loader.get_template('brTalkingformula/pilotos.html')
    items = Piloto.objects.order_by('nome')[0:]
    context = {
        'pilotos':items
    }

    return HttpResponse(template.render(context, request))
```

4.3.3 `def pilotosDetails` — `def circuitoDetails` — `def corridaDetails`

Para todas estas funções o conceito foi de dar *display* das características/detalhes de um objeto específico relativo a um dos modelos.
Tomando o exemplo de `def pilotosDetails`, a função carrega o template do .html e usa a função `try:` para caso os detalhes do piloto escolhido (`pilotoid`) não existam, para devolver a excepção a `"Piloto does not exist"` (`pilotos`) a serem mostrados. Por fim a função terá de dar `return` destes dois elementos.

```
def pilotosDetails(request, pilotoid):
    template = loader.get_template('brTalkingformula/
       piloto_detalhes.html')

    try:
        piloto = Piloto.objects.get(pilotoid = pilotoid)
        context = {
            'piloto' : piloto
            }
    except Piloto.DoesNotExist:
        raise Http404("Piloto does not exist")
    return HttpResponse(template.render(context, request))
```

Para definir qual o objeto de que queremos analisar os detalhes, adicionámos uma variável à função, sendo no caso do `def pilotosDetails` a variável `pilotoid`. Esta variavel é definida pelo url sendo tal pormenorizado no `urls.py`:

```
urlpatterns = [
#...
    path('pilotos/<int:pilotoid>/', views.pilotosDetails, name='
       pilotosDetails'),
#...
]
```

No .html, dependendo do url que é mostrado no página, irá levar ao respetivo piloto. Por exemplo `'/brTalkingformula/pilotos/8/'` leva-nos à página que exibe os detalhes do piloto com `'pilotoid'` = 8.

4.3.4 def resultadosPilotos

A função `def resultadosPilotos` é responsável por mostrar a página dedicada aos *standings* dos pilotos. Semelhante ao que é feito em 4.3.1, mas com a complexidade extra de se poder ver os resultados de todas as épocas (registadas).

```
def resultadosPilotos(request, epoca_ano=Epoca.objects.first().
   ano):
    template = loader.get_template('brTalkingformula/resultados.
       html')
    epocas = Epoca.objects.all()
    standings = calc_resultadosPilotos(epoca_ano)

    context = {
        'standings': standings,
        'epocas' : epocas,
    }

    return HttpResponse(template.render(context, request))
```

Para definir a época em que queremos analisar os standings/resultados, adicionámos a variável `epoca_ano` que decide qual a época a mostrar os resultados. Por *default*, são mostrados os standings da última época. No .html são mostrados todas as épocas, por meio da variável `'epocas'`, sendo que hiperligação ao clicar numa das épocas apresentadas nessa página leva a uma nova página com os standings da época selecionada. Tal é mediado no `urls.py`:

```
    urlpatterns = [
    #...
```

```
3         path('resultados/', views.resultadosPilotos, name='
              ultimos_resultados'),
4         path('resultados/<int:epoca_ano>', views.
              resultadosPilotos, name='resultados'),
5     #...
6     ]
```

O path na linha 3, é relativo ao path que o botão resultados na navbar devolve, o que corresponde à ultima época. Já o path na linha abaixo, na linha 4, corresponde ao path do ano que é selecionado. Por exemplo, no .html é gerado um url da forma `'brTalkingformula/resultados/2021'`, pelo que `'epoca_ano'=2021` e consequentemente o views.py irá dar return dos standings correspondentes a essa época.

4.3.5 def noticiasDetails

Esta função `'views.py'` é na sua base idêntica às explicadas em 4.3.3, no entanto adicionámos a capacidade de fazer comentários, sendo esse aspeto que iremos discutir agora.

Primeiramente, é importante lembrar relembrar que no diagrama ER 3, à entidade 'Noticia' está relacionada a entidade 'Comentario', sendo que para uma notícia poderão haver vários comentários, sendo que este último não existe o primeiro.

Voltando à `'views.py'` este é o aspeto da função:

```
1  def noticiasDetails(request, noticiaid):
2      template = loader.get_template('brTalkingformula/
           noticia_detalhes.html')
3      noticia = Noticia.objects.get(noticiaid = noticiaid)
4
5      comentarios = Comentario.objects.filter(noticia_noticiaid =
           noticiaid)
6      novo_comentario = None
7
8      #Quando postado um novo comentario
9      if request.method == 'POST':
10         comentario_form = ComentarioForm(data=request.POST)
11         if comentario_form.is_valid():
12
13             novo_comentario = comentario_form.save(commit=False)
14             novo_comentario.noticia_noticiaid = noticia
15             novo_comentario.save()
```

```
    else:
        comentario_form = ComentarioForm()

    context = {
        'noticia' : noticia,
        'comentarios': comentarios,
        'novo_comentario' : novo_comentario,
        'comentario_form' : comentario_form
              }
    return HttpResponse(template.render(context, request))
```

A variável `comentarios` obtém todos os comentários relativos à noticia selecionada. De seguida, fizemos uso do `forms.py` que é responsável por gerar o campo para completar as nossas variáveis:

```
from .models import Comentario
from .models import Noticia
from django import forms

class ComentarioForm(forms.ModelForm):
    class Meta:
        model = Comentario
        fields = ('nome', 'username', 'corpo')
```

Queremos completar o 'Nome', 'username' (para diferenciar caso hajam nomes iguais) e o 'corpo' da mensagem. A data de criação do comentário é definido no modelo do comentário para corresponder à data atual: `criadoem = models.DateTimeField(auto_now_add=True)`.

Voltando novamente ao `views.py`, segue o código para gerar o novo comentário.

```
novo_comentario = None

if request.method == 'POST':
        comentario_form = ComentarioForm(data=request.POST)
        if comentario_form.is_valid():

            novo_comentario = comentario_form.save(commit=False)
            novo_comentario.noticia_noticiaid = noticia
            novo_comentario.save()
    else:
        comentario_form = ComentarioForm()
```

No lado do .html, quando é feito um request de `POST`, a variável `comentario_form` irá guardar a

informação enviada pelo formulário. O método do Django `.is_valid()`, permite verificar se os campos foram corretamente completados.

Assim, se o formulário completado for validado, primeiro é utilizado o método `.save()` para associar à variável `novo_comentario` mas com `commit=False` para impedir que a variável seja armazenada diretamente na base dados pois ainda é necessário associar à noticia correta o `'novo_comentario'`. Utilizando o comando `novo_comentario.noticia_noticiaid = noticia`, tal é alcançado e por fim podemos novamente utilizar o método `.save()` para guardar na base da dados.

Se for um pedido `GET` (contrário do POST), é gerado o bloco para preencher o formulário.

4.4 admin.py

Importamos os modelos do models.py e utilizamos a função *admin.site.register* para registar um modelo com o site do admin do Django. Isto permite-nos usar o site do admin para criar, modificar ou apagar dados do projeto.

```
1  from .models import Circuito
2  from .models import Corrida
3  from .models import Piloto
4  from .models import PilotoEquipa
5  from .models import Equipa
6  from .models import Construtor
7  from .models import Resultados
8  from .models import Comentario
9  from .models import Epoca
10 from .models import Noticia
11
12 admin.site.register(Circuito)
13 admin.site.register(Corrida)
14 admin.site.register(Piloto)
15 admin.site.register(PilotoEquipa)
16 admin.site.register(Equipa)
17 admin.site.register(Construtor)
18 admin.site.register(Resultados)
19 admin.site.register(Comentario)
20 admin.site.register(Epoca)
21 admin.site.register(Noticia)
```

Na figura 4, é possível ver a página do admin, após fazer o registo como admin.

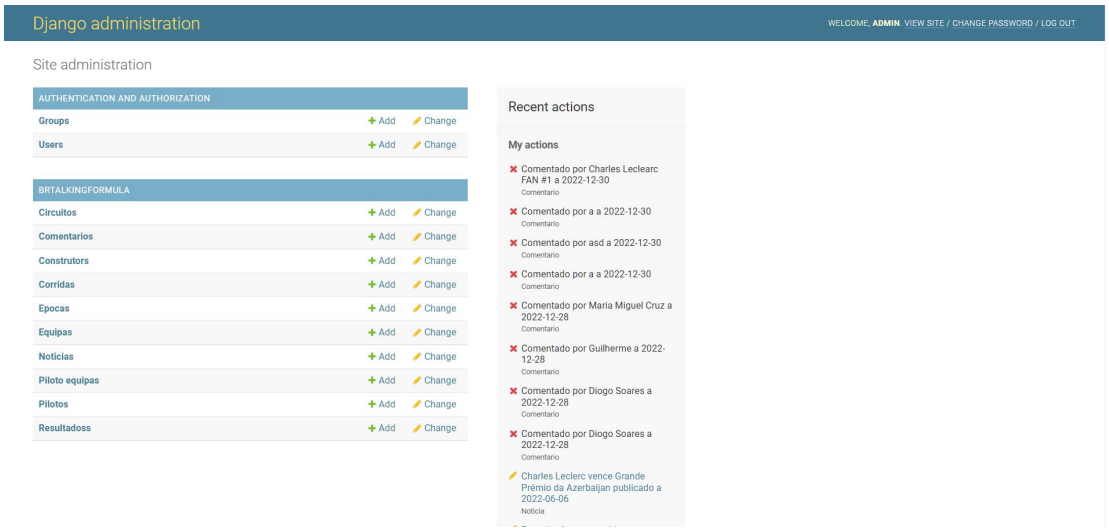

Figura 4: Print da página do admin, após fazer o registo como admin

Vamos agora fazer um *walkthrough* para adicionar um **piloto** através do admin do django. Após carregar no ícone **ADD** nos pilotos, temos a página que está representada na figura 5.

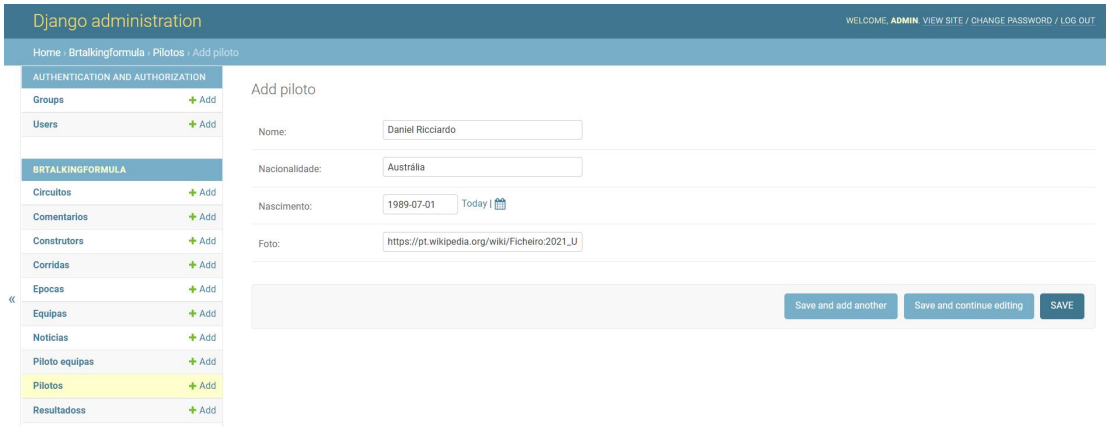

Figura 5: Print da página do admin, após preencher os dados do piloto

De seguida, preenchemos os dados do piloto e carregamos em **SAVE**.
Como podemos ver na figura 6, o piloto **Daniel Ricciardo** foi adicionado com sucesso:

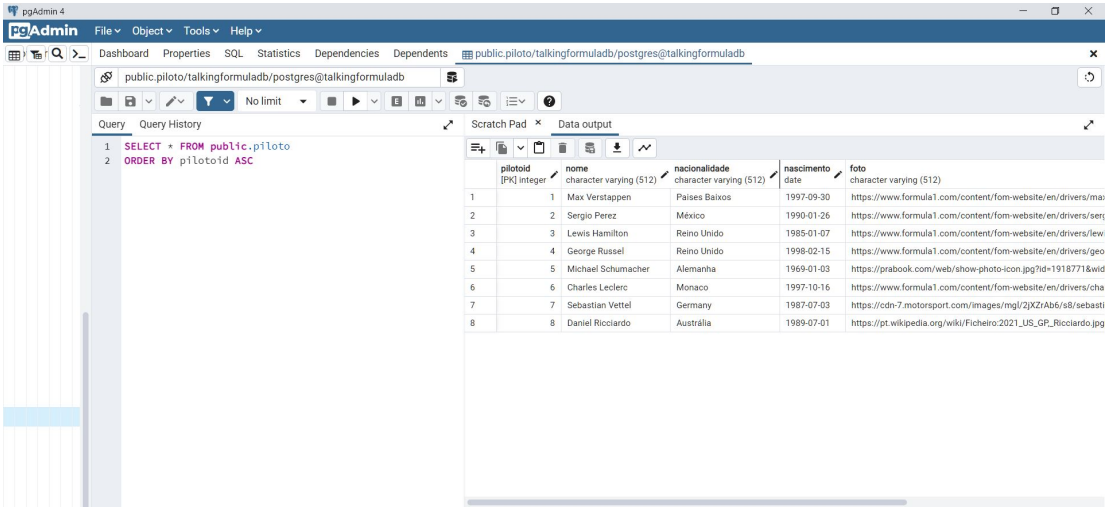

Figura 6: Confirmação dos dados no pgAdmin

5 Html e CSS

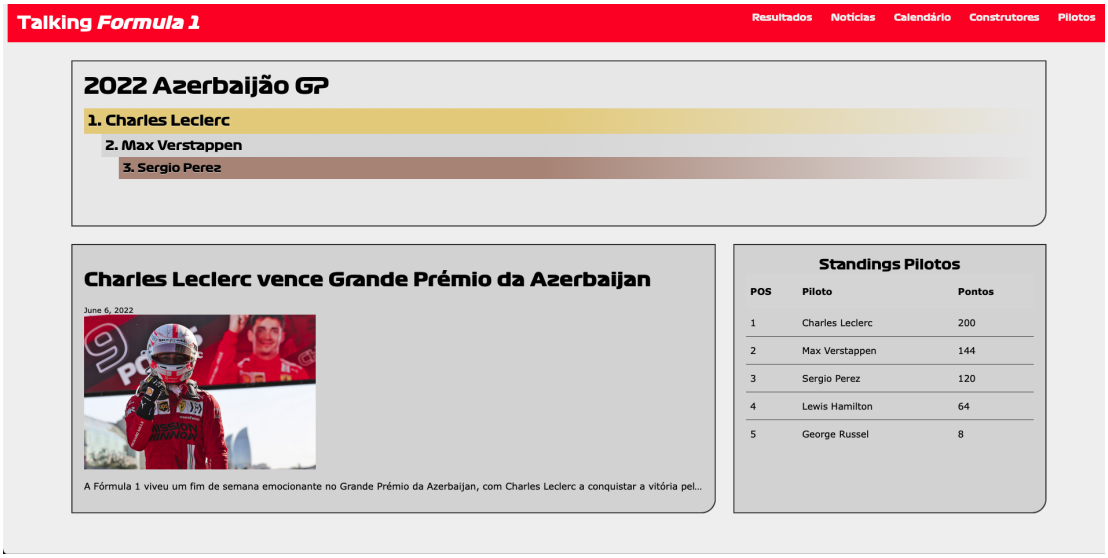

Figura 7: Para esta página foi gerado um `.wrapper` com 3 colunas e com rows com a profundidade definida para se adaptar ao conteúdo. Foram também adicionados alguns detalhes estilísticos como nos resultados, onde o primeiro classificado tem o fundo com o gradiente em dourado. A vasta maioria dos elementos tem uma hiperligação no *website* associada de forma a completar a informação que é mostrada nesta página.

Para aceder de uma forma prática, criámos uma navbar que funciona como index para as variadas entidades criadas neste projeto.
Em `CSS` criámos a classe `.topnav` que é responsável pela estilização desta divisória.

```
.topnav {
    background-color: #FF1801;
    overflow: hidden;
    font-family: formulaonefont;

    position: fixed;
    top: 0;
    width: 98.7%;
  }
```

Além disso adicionámos a *font* da Fórmula 1 para aumentar a complexidade do nosso *design*:

```
    @font-face {
    font-family: formulaonefont;
    src: url(fonts/Formula1-Bold.ttf);
}
```

Também adicionámos às diferentes divisórias, subclasses `:hover` para quando colocando o rato sob essas divisórias haver alterações no seu estilo (por exemplo para modificar a cor da divisória o que

ajuda a dar enfâse):

```
    .pos1:hover{
  background-color: #ebd57d;
  transition: 1s;
}
```

Para criar os blocos (com o resultado da última corrida, última notícia e resultados da última/corrente época), partimos da class `.wrapper` para divisórias, sendo que gera o esqueleto de uma grid. Depois gerámos subclasses com as características dos blocos que queríamos:

```
    .top_block {
  grid-column: 1 / 4;

  padding: 20px;
  background-color: rgba(208, 208, 208, 0.179);


  border: 2px solid rgba(0, 0, 0, 0.784);
  border-bottom-right-radius: 25px;
}
```

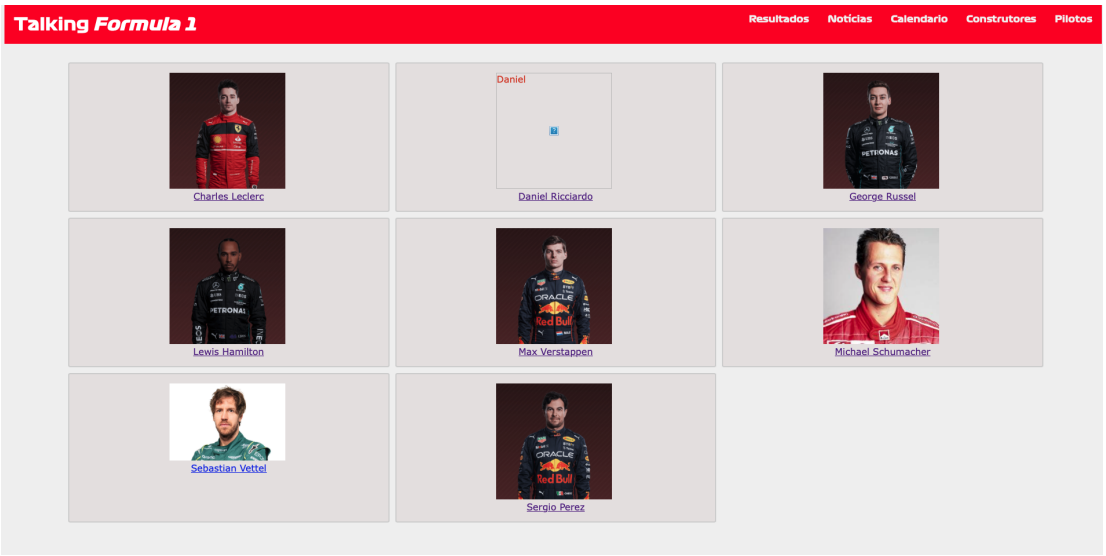

Figura 8: Utilização do .wrapper para dar display dos pilotos nas boxs.

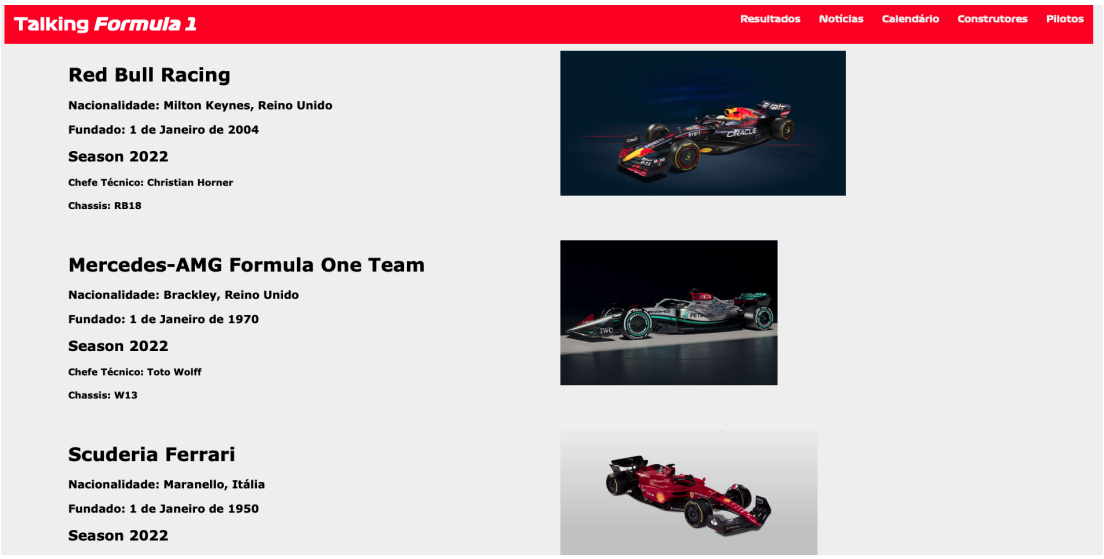

Figura 9: Utilização do .wrapper para dar display dos construtores. São criadas duas colunas e com o tamanho das rows adaptadas ao conteúdo.

A nível do `.html`, utilizámos o `<div>` para exibir grande parte dos nossos dados e também como base que foi estilizada pelo CSS que mencionámos. Também criámos hiperligações, `<a>`, para grande parte dos elementos de forma a completar e direcionar para a informação existente na nossa base de dados. Para mostrar vários objetos (de uma dado modelo) enviados em lista, utilizamos ciclos `for`.
Geramos tabelas para mostrar os dados do calendário, como podemos ver na figura 10e dos resultados, como se vê na fig11.

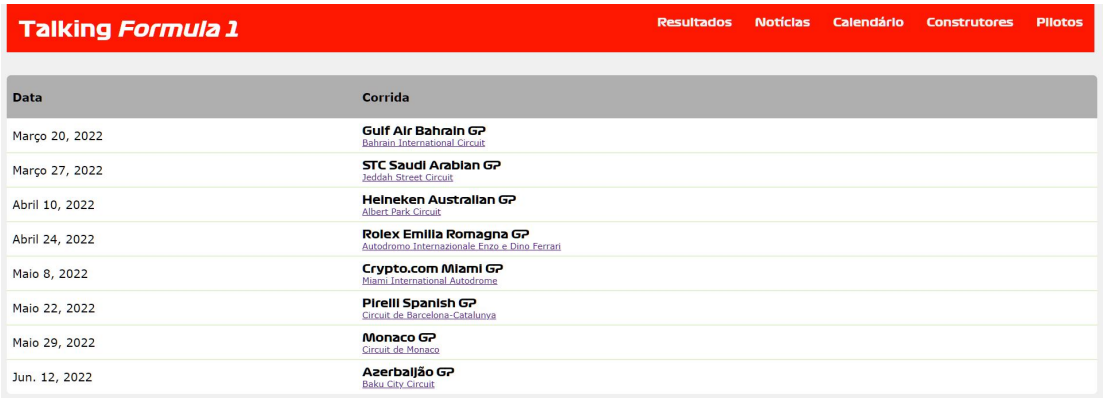

Figura 10: Print do calendário com as corridas e os respetivos circuitos

POS	Piloto	Nacionalidade	Construtor	Pontos
1	Charles Leclerc	Monaco	Scuderia Ferrari	200
2	Max Verstappen	Paises Baixos	Red Bull Racing	144
3	Sergio Perez	México	Red Bull Racing	120
4	Lewis Hamilton	Reino Unido	Mercedes-AMG Formula One Team	64
5	George Russel	Reino Unido	Mercedes-AMG Formula One Team	8

Figura 11: Print dos resultados

Falando do separador calendário, temos o seguinte `for cycle` para percorrer os dados da **Data** e da **Corrida**, que engloba o circuito e a corrida. No excerto de código abaixo, temos o tal `for cycle`.

```
{% for corridaROW in corridas %}
                    <tr>
                        <td>{{corridaROW.ocorreem}}</td>
                        <td>
                            <div class="gp">
                                <a id="logo" href="/
                                   brTalkingformula/corrida/{{
                                   corridaROW.ronda}}">{{
                                   corridaROW.nome }}</a>
                            </div>
                        <div class="track"> <a href ='/
                           brTalkingformula/circuitos/{{corridaROW
                           .circuito_circuitoid.circuitoid}}/'>{{
                           corridaROW.circuito_circuitoid.nome}}</
                           a> </div>
                        </td>
                    </tr>
                    {% endfor %}
```

Passando para os resultados, fizemos também um `for cycle`, que percorre a **posição, piloto, nacionalidade, construtor e, por fim, pontos**.

```
                      {% for standingROW in standings %}
                    <tr>
                        <td> {{ forloop.counter }}</td>
                        {% for item in standingROW %}
                            {% with counter=forloop.counter0
                               %}
```

```
                                    {% if counter == 0 %}
                                        <td>{{ item.
                                            piloto_pilotoid.nome }}
                                            </td>
                                        <td>{{ item.
                                            piloto_pilotoid.
                                            nacionalidade }}</td>
                                        <td>{{ item.
                                            equipa_equipaid.
                                            construtor_construtorid
                                            .nome }}</td>
                                    {% elif counter == 1 %}
                                        <td class=>{{ item }}</td
                                            >
                                    {% endif %}
                                {% endwith %}
                                {% endfor %}
                            </tr>
                            {% endfor %}
```

Relativamente aos comentários, para mostrar todos os comentários usamos um ciclo `for`.

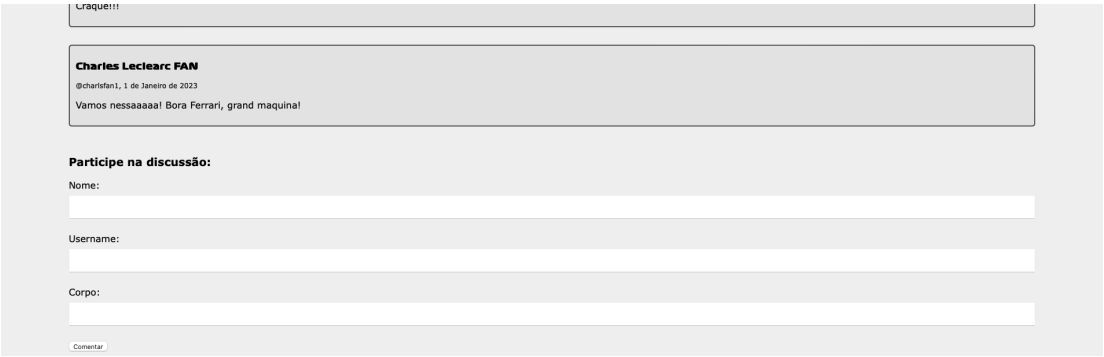

Figura 12: Form para criar um comentário no .html dos detalhes de uma notícia

Mais importante é o código do formulário:

```
    <div>
    {% if novo_comentario %}
    <div>
        Boas discuss es!
    </div>
    {% else %}
    <h3>Participe na discuss o:</h3>
    <form method="post" style="margin-top: 5px;">
```

```
9         {{ comentario_form.as_p }}
10        {% csrf_token %}
11        <button type="submit">Comentar</button>
12      </form>
13      {% endif %}
14    </div>
```

Caso ainda não tenha sido completado o form para criar um novo comentário, é aberto o elemento `<form>` que em Django, o método 'POST' permitirá receber os dados no nosso `views.py`.

Dentro do `<form>` temos a aplicação do método `.as_p` que dá render dos elementos do objeto `comentario_form` numa sequência de parágrafos. Por fim, `% csrf_token %` é um elemento de segurança para proteger o website. O `<button>` com o atributo `submit` irá dar *submit* do form preenchido.