



FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Relatório Meta 2 Projeto

Base de Dados

Projeto realizado por:

Ana Raquel Ferreira	nº 2017248936
José Veríssimo Lamas	nº2017259895
Ricardo Mendes Figueiredo	nº 2017254736

- Divisão de tarefas

<i>Elementos do grupo</i>	<i>Ana Raquel</i>	<i>José Lamas</i>	<i>Ricardo Figueiredo</i>
<i>Protótipos</i>	Login, Registo e Criar Torneio	Lista Torneios, Informação Equipa e Informação Torneio	Perfil, Lista de Equipas e Inscrever em Equipas
<i>Tabelas Base de Dados</i>	- auth_user - accounts_users	- teams_player - teams_position - teams_team - teams_teamplayer	- tournaments_gamefields - tournaments_games - tournaments_manager - tournaments_playergoals - tournaments_tournaments - tournaments_tournamentslots
<i>Páginas</i>	Login Registo Homepage Perfil Pessoas Editar Perfil	Lista de equipas Lista de equipas de um utilizador Criar Equipa Detalhes da Equipa Área do Capitão de Equipa	Criar torneio Detalhes de um torneio Datas proibidas de um torneio Lista de torneios

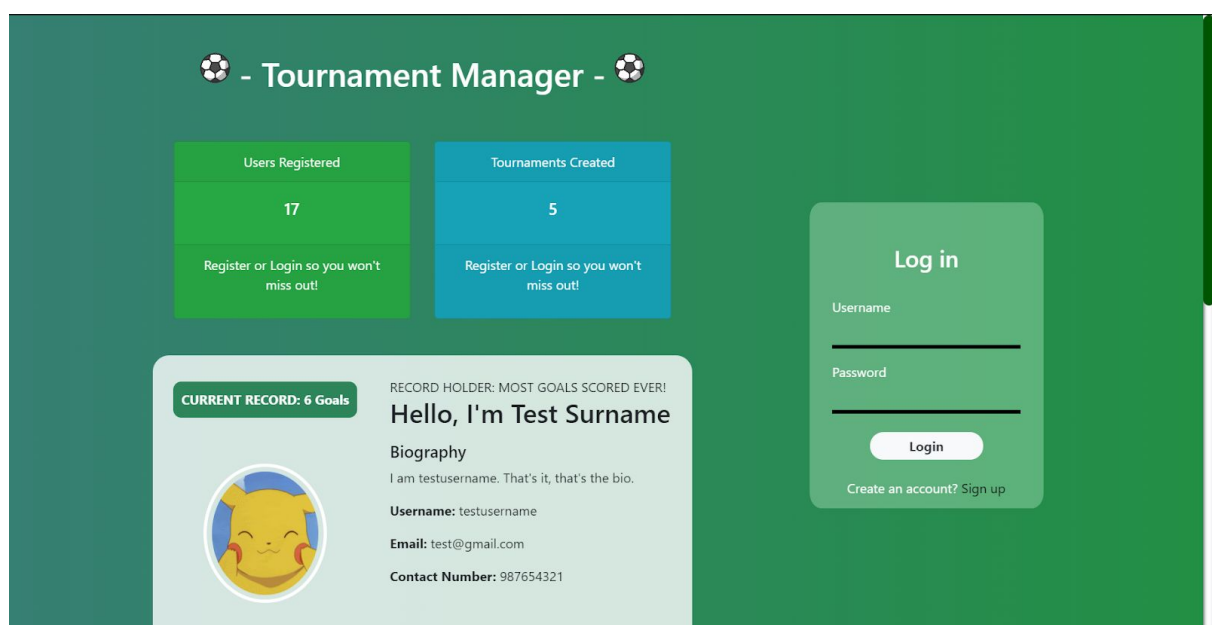
Geral: Foi usada MySQL.

- Secções de trabalho


Ana Raquel

--- Ecrãs desenvolvidos ---

Login:



Registro:



Welcome

If you already have an account, please Login

Login

Register to play

<input type="text"/>	<input type="text"/>
<input type="password"/>	<input type="text"/>
<input type="password"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>

Register

Página Inicial:

Tournament Manager

HomeTournamentsPeopleMy Profile

Logout

Users Registered

17

Click here to see all users

All Users

Tournaments Created

5

Check out all the tournaments here

All Tournaments

Teams Created

9

See all the teams here!

All Teams

Managers

2

Check out our awesome managers!

All Managers

Number Of Games Played

0

Check out the results here

See Tournaments

Number Of Games Planned

2

See these games here!

See Tournaments

Total Goals Scored

8

Help improve this statistic!
Sign up for teams here

Teams

Teams Looking For Players

9

Click here to register in one!

See Teams

Hello, Test Surname

ManagerPlayer

Email: test@gmail.com

Contact: 987654321

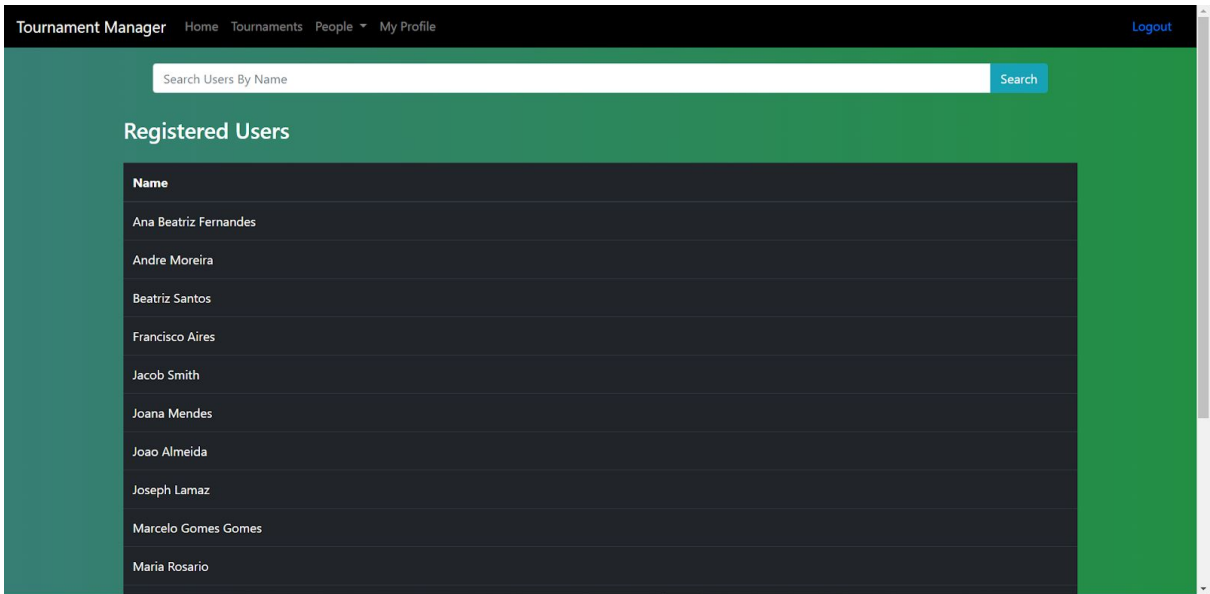
Saldo: 100€

Edit Profile

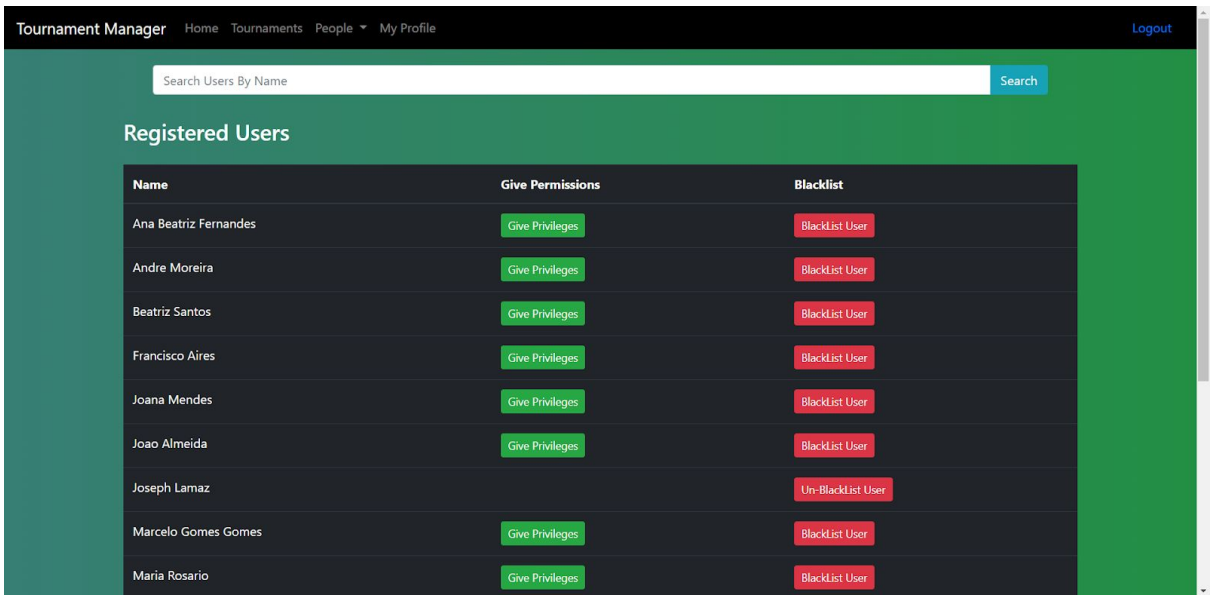
CURRENT RECORD: 6 Goals

RECORD HOLDER: MOST GOALS SCORED EVER!
Hello, I'm Test Surname

Ecrã de pessoas (Utilizadores registados, visto por utilizador normal):



Ecrã de pessoas (Utilizadores registados, visto por um administrador):




Perfil de utilizador (ver o próprio perfil):

Tournament Manager

HomeTournamentsPeople▼My Profile

Logout



Edit Profile

Manage Balance

ABOUT ME

Hello, I'm Test Surname

Manager

Player

Biography

I am testusername. That's it, that's the bio.

Username: testusername

Date Joined: Nov. 21, 2019, 6:32 p.m.

Email: test@gmail.com

Last Login: Nov. 30, 2019, 6:28 p.m.

Contact Number: 987654321

CC: 12345678

Player Area

Status: Good!

Saldo: 100€


Managed Tournaments

Perfil de utilizador (ver o de outra pessoa):

Tournament Manager

HomeTournamentsPeople▼My Profile

Logout



ABOUT ME

Hello, I'm Raquel Ferreira

Biography

I am Raquel. My mom made me be here.

Username: raquel

Date Joined: Nov. 24, 2019, 6:12 p.m.

Email: rachlogan98@gmail.com

Last Login: Nov. 29, 2019, 9:10 p.m.

Contact Number: 917064358

Editar Perfil:

The screenshot shows the 'Edit Profile' form within the 'Tournament Manager' application. The form is set against a dark green background. It includes a navigation bar at the top with links to 'Home', 'Tournaments', 'People', and 'My Profile', and a 'Logout' button. The form itself has a title 'Edit Profile' and contains three main sections: 'Phone Number' with a text input field containing '987654321', 'Profile Picture' with an 'Upload a file' button, and 'Biography' with a text area containing 'I am testusername. That's it, that's the bio.' At the bottom of the form are two buttons: 'Save' and 'Back to Profile'.

NOTA: Não estão presentes todos os elementos das páginas, devido à presença da scroll bar.

--- Validações ---

Na framework Django, tabelas da base de dados são usadas na forma de modelos.

O modelo Users representa a tabela `accounts_users` da base de dados, que estende a tabela `auth_user` criada pela framework Django. Esta tabela contém o username, password, email, nome e informações da data de registo e última vez que o utilizador realizou o login na plataforma. O referido modelo Users e o código MySQL equivalente encontram-se nas imagens abaixo.

```
""" EQUIVALENT SQL """
# CREATE TABLE accounts_users (
#     FOREIGN KEY(user)
#     REFERENCES auth_user(id)
#     ON DELETE CASCADE,
#     cc BIGINT,
#     PRIMARY KEY(cc),
#     phoneNumber BIGINT NOT NULL,
#     hasPriv boolean NOT NULL,
#     image varchar(100) NOT NULL,
#     bio varchar(250),
# );
```

```

# Table users to store all app users
class Users(models.Model):
    """
    auth_user contains:
    - username
    - password
    - email
    - first name
    - last name
    - ...
    """
    user = models.OneToOneField(User,
                                related_name="user_extended", # reference this model from the auth_user model
                                on_delete=models.CASCADE,
                                unique=True) # reference to the django user model
    cc = models.BigIntegerField(primary_key=True, validators=[MinValueValidator(0)]) # CC is the primary key
    phoneNumber = models.BigIntegerField(validators=[
        RegexValidator(
            regex=r'^\+?1?\d{9,15}$',
            message='Phone number must be at least 9 digits and maximum 15',
            code='invalid_number'
        ),
    ])
    hasPriv = models.BooleanField() # whether the user has privileges or not (create tournaments, ...)
    image = models.ImageField(default='profile-pics/default.png', upload_to='profile-pics') # store the profile picture
    bio = models.TextField(max_length=250, default="", blank=True) # let the user describe themselves

    def __str__(self):
        return f'{self.user.username} Profile'

```

Este modelo possui validações dos seus parâmetros:

- Validação que o número do cartão de cidadão é um número inteiro positivo (MinValueValidator(0))
- Validação do número de contacto com uma expressão regex: “`^\+?1?\d{9,15}$`”. Significa que permite a utilização opcional do indicativo do país (com sinal ‘+’) e um número mínimo de dígitos igual a 9 assim como um número máximo de dígitos igual a 15, permitindo assim a maior diversidade de contactos possíveis válidos. Caracteres ‘^’ e ‘\$’ garantem que não existem apenas correspondências parciais, é obrigatório a correspondência total do input.
- Validação do comprimento máximo da biografia do utilizador: 250 caracteres.

O registo de utilizadores é ainda validado no seu form, representado abaixo.

```

class registerForm(forms.Form):
    username = forms.CharField(label='Username', max_length=50,
                               widget=forms.TextInput(attrs={'placeholder': '', 'class': 'formInput'}),
                               validators=[RegexValidator(r'^[A-Za-z0-9]+(?:[._-][A-Za-z0-9]+)*$',
                                                         "Username must only contain letters, hyphens, dots, "
                                                         "underscores and numbers. Username can't end with a special "
                                                         "character.", 'Invalid username')])
    password = forms.CharField(widget=forms.PasswordInput(attrs={'placeholder': '', 'class': 'formInput'}))
    confPassword = forms.CharField(label="Confirm Password",
                                   widget=forms.PasswordInput(attrs={'placeholder': '', 'class': 'formInput'}))
    email = forms.EmailField(widget=forms.EmailInput(attrs={'placeholder': '', 'class': 'formInput'}))
    firstName = forms.CharField(label='First Name', max_length=50,
                                widget=forms.TextInput(attrs={'placeholder': '', 'class': 'formInput'}))
    lastName = forms.CharField(label='Last Name', max_length=50,
                               widget=forms.TextInput(attrs={'placeholder': '', 'class': 'formInput'}))
    # added for the accounts_users table info
    contactNumber = forms.IntegerField(label='',
                                       widget=forms.TextInput(attrs={'placeholder': '', 'class': 'formInput'}),
                                       validators=[
                                           RegexValidator(
                                               regex=r'^\+?1?\d{9,15}$',
                                               message='Phone number must be at least 9 digits and maximum 15',
                                               code='invalid_number')
                                       ])
    # added for the accounts_users table info
    cc = forms.IntegerField(label='Citizen Number',
                           widget=forms.TextInput(attrs={'placeholder': '', 'class': 'formInput'}),
                           validators=[RegexValidator(r'\d{8,8}', 'CC must be 8 digits', 'Invalid CC number')])

```

O form de registo contém validações não só para a tabela `accounts_users`, como para a tabela `auth_user`:

- Validação do username com uma expressão regex: `^[A-Za-z0-9]+(?:[._-][A-Za-z0-9]+)*$`. Significa que o username apenas pode conter letras (maiúsculas e minúsculas), números e opcionalmente os caracteres `'.'`, `'_'` e `'-'`. Estes caracteres não podem estar no fim do username e mais uma vez a correspondência não pode ser parcial, é correspondência total do input.
- Validação do número de cartão de cidadão (cc) com uma expressão de regex: `\d{8,8}`. Significa que o número de cartão de cidadão tem de ser exatamente 8 dígitos.
- Validação do número máximo de caracteres do username, primeiro nome e último nome: máximo de 50 caracteres cada.

Estas validações não bastam, pois ainda é preciso validar a confirmação da password inserida e os dados apesar de válidos podem já estar presentes na base de dados, por isso as funções de validação do form são ainda usadas.


```

def clean_password2(self):
    password1 = self.cleaned_data.get("password")
    password2 = self.cleaned_data.get("confPassword")
    if password1 and password2 and password1 != password2:
        return False
    return True

# check if username is already in the database, returns false in case it is not "safe"
def clean_username2(self):
    username = self.cleaned_data.get("username")
    """ EQUIVALENT SQL """
    # SELECT count(*) FROM 'auth_user' WHERE auth_user.username = username
    if User.objects.filter(username=username).count() != 0:
        return False
    return True

# check if cc is already in use, returns false in case it is not "safe"
def clean_cc2(self):
    cc = self.cleaned_data.get("cc")
    """ EQUIVALENT SQL """
    # SELECT count(*) FROM 'accounts_users' WHERE accounts_user.cc = cc
    if Users.objects.filter(cc=cc).count() != 0:
        return False
    return True

# check if email is already in use, returns false in case it is not "safe"
def clean_email2(self):
    email = self.cleaned_data.get("email")
    """ EQUIVALENT SQL """
    # SELECT count(*) FROM 'auth_user' WHERE auth_user.email = email
    if User.objects.filter(email=email).count() != 0:
        return False
    return True

```

- Clean_password2: verifica se os dois campos de password são iguais
- Clean_username2: verifica se o username já se encontra na base de dados
- Clean_cc2: verifica se o número de cartão de cidadão já se encontra na base de dados
- Clean_email2: verifica se o email já se encontra na base de dados

--- Tratamento de erros ---

Se a informação inserida nos forms de registo, login, editar perfil ou mudança da quantidade de saldo na conta não forem válidas, uma mensagem de erro é mostrada ao utilizador. Nada é inserido na base de dados e é pedido ao utilizador a reinserção de dados no form correspondente. Por exemplo, no tratamento do form de registo, o código seguinte é efetuado.

```
if request.method == "POST":
    form = registerForm(request.POST)
    if form.is_valid() \
        and form.clean_username2() \
        and form.clean_password2() \
        and form.clean_email2() \
        and form.clean_cc2():...

    else:
        # return all the user messages that apply at once
        if not form.clean_username2():
            messages.error(request, "Username already in use. Please choose another one.")
        if not form.clean_password2():
            messages.error(request, "Passwords don't match")
        if not form.clean_email2():
            messages.error(request, "Email is already registered. Log in or try another one.")
        if not form.clean_cc2():
            messages.error(request, "CC is already registered. Identity theft is a real crime, use your own.")

content = {'form': form}
return render(request, 'registration/register.html', content)
```

São realizadas as funções de validação e dependendo dos erros, são mostradas as mensagens de erro ao utilizador.

--- Estruturação do código ---

Na framework Django são usadas aplicações para facilitar a divisão do trabalho. Contém as aplicações Tournament Manager (junta o trabalho de todos) assim como as aplicações individuais que representam o trabalho de cada elemento do grupo. O meu trabalho encontra-se na aplicação “accounts”, apesar de algumas definições estarem presentes fora pois é necessário a coordenação e coerência entre todo o trabalho.

Na pasta denominada accounts, encontram-se as diretorias: migrations, static e templates.

A pasta migrations contém os ficheiros de migração para a base de dados, é assim que são feitas alterações na base de dados com a alteração dos modelos de Django. A pasta static contém os ficheiros de css utilizados para o design da aplicação. Por fim, a pasta templates contém os ficheiros html utilizados na aplicação.

Dentro da pasta accounts, podemos encontrar os ficheiros:

- forms.py
- models.py
- urls.py
- views.py

O ficheiro forms.py contém os forms utilizados para a inserção de dados do utilizador. O ficheiro models.py contém os modelos utilizados na aplicação, neste caso o modelo Users. O ficheiro urls.py

contém a informação para redirecionar o utilizador (quais urls existem, como referenciar um certo url e a sua view correspondente). Por fim o ficheiro views.py contém a lógica back-end de todas as páginas da aplicação. A parte mais importante da aplicação situa-se neste último ficheiro.

```
def loginView(request):...

# username admin is the admin
def registerView(request):...

@login_required
def logoutView(request):...

@login_required
def dashboardView(request):...

@login_required
def profileView(request, username):...

@login_required()
def peopleView(request):...

@login_required()
def peopleSearch(request, filterPeople):...

@login_required()
def editProfileView(request):...
```

- loginView contém a lógica da página de login, a primeira página vista pelo utilizador se não estiver “logged in”
- registerView contém a lógica da página de registo, onde o utilizador se pode registar na plataforma
- logoutView contém a lógica do botão de Logout presente em todos os ecrãs, faz o logout do utilizador que fez login e redireciona para a página inicial, neste caso será a página de login.
- dashboardView contém a lógica para a página inicial do utilizador que tem o login efetuado. É a página default que o utilizador vê, exceto se não tiver feito o login ou efetuou o logout.
- profileView contém a lógica da página do perfil de todos os utilizadores, controlando a quantidade de informação mostrada, dependente se é o próprio perfil ou não. Esta distinção foi realizada pois a partilha de certas informações pessoais pode ser um risco de segurança, assim como o controlo do que cada pessoa pode fazer no perfil. Contém também a lógica do controlo de saldo da própria conta.
- peopleView contém a lógica da página que lista todos os utilizadores da plataforma
- peopleSearch contém a lógica da pesquisa de pessoas específicas. Esta função é utilizada quando um utilizador pesquisa pelo nome de alguém que quer encontrar na plataforma ou quando o utilizador quer ver apenas quem são os managers presentes na plataforma
- editProfileView contém a lógica da página de editar o seu próprio perfil de utilizador na plataforma.

--- SQL relevante em cada ecrã ---

Login:

```
# STATS
# SELECT COUNT(accounts_users.cc) FROM accounts_users, auth_user WHERE accounts_users.user_id = auth_user.id AND NOT auth_user.is_superuser
numberUsers = Users.objects.filter(user__is_superuser=False).count()
numberTour = Tournaments.objects.all().count() # SELECT COUNT(*) FROM 'tournaments_tournaments'
tournamentsReference = Tournaments.objects.all() # SELECT * FROM 'tournaments_tournaments'

""" EQUIVALENT SQL """
# SELECT player_id,sum(goals) as total FROM tournaments_playergoals
# GROUP BY player_id
# ORDER by total DESC
# LIMIT 1
recordId = PlayerGoals.objects.order_by('-total').values('player').annotate(total=Sum('goals'))[0]
recordPlayerID = recordId['player']
recordGoals = recordId['total']

# SELECT * FROM 'accounts_users' WHERE cc = recordPlayerID
recordUser = Users.objects.filter(cc=recordPlayerID).first()

""" SQL EQUIVALENT """
# SELECT * FROM tournaments_games, tournaments_tournamentslots
# WHERE tournaments_games.slot_id = tournaments_tournamentslots.id
# AND tournaments_tournamentslots.date <= DATE_ADD(NOW(), INTERVAL 7 DAY)
gamesNextWeek = Games.objects.filter(slot__date__lte=datetime.now().date()+timedelta(days=7))
```

- Estatística do número de utilizadores na plataforma
- Estatística do número de torneios criados na plataforma
- Torneios criados e a sua informação
- Jogador que tem feito mais golos ao longo do tempo
- Informações sobre jogos na próxima semana

Registo:

```
# save the user in the auth_user table
""" SQL EQUIVALENT """
# INSERT INTO 'auth_user' ('id', 'password', 'last_login', 'is_superuser', 'username', 'first_name',
# 'last_name', 'email', 'is_staff', 'is_active', 'date_joined') VALUES (NULL, '', NULL, '0', 'username',
# 'firstName', 'lastName', 'email', '0', '1', '')
authUser = User(username=username, email=email, first_name=firstName, last_name=lastName)
""" SQL EQUIVALENT """
# UPDATE 'auth_user' SET 'password' = password WHERE 'auth_user'.id = authUser.id
authUser.set_password(raw_password=password)
authUser.save()

# save the user to the accounts_users table
if username == "admin":
    """ SQL EQUIVALENT """
    # INSERT INTO 'accounts_users' ('cc', 'hasPriv', 'user_id', 'phoneNumber') VALUES (cc, '1', authUser.id,
    # contactNumber)
    user = Users(user=authUser, phoneNumber=contactNumber, cc=cc, hasPriv=True)
    user.save()
else:
    """ SQL EQUIVALENT """
    # INSERT INTO 'accounts_users' ('cc', 'hasPriv', 'user_id', 'phoneNumber') VALUES (cc, '0', authUser.id,
    # contactNumber)
    user = Users(user=authUser, phoneNumber=contactNumber, cc=cc, hasPriv=False)
    user.save()
```

- Adicionar utilizadores na tabela de auth_user e na tabela de accounts_users

Dashboard:

```
""" SQL EQUIVALENT """
# SELECT COUNT(*) FROM tournaments_games, tournaments_tournamentslots
# WHERE tournaments_games.slot_id = tournaments_tournamentslots.id
# AND tournaments_tournamentslots.date < DATE(NOW())
gamesPlayed = Games.objects.filter(slot__date__lt=datetime.now().date()).count()
""" SQL EQUIVALENT """
# SELECT COUNT(*) FROM tournaments_games, tournaments_tournamentslots
# WHERE tournaments_games.slot_id = tournaments_tournamentslots.id
# AND tournaments_tournamentslots.date > DATE(NOW())
gamesFuture = Games.objects.filter(slot__date__gt=datetime.now().date()).count()
```

- Estatística do número de jogos que já decorreram
- Estatística do número de jogos planeados para o futuro

Perfil:

```
""" EQUIVALENT SQL """
# SELECT COUNT(accounts_users.cc)
# FROM tournaments_manager, accounts_users, auth_user
# WHERE tournaments_manager.user_id = accounts_users.cc
# AND accounts_users.user_id = auth_user.id
# AND auth_user.id = userProfile.id
if Manager.objects.filter(user_id=userProfile.user_extended.cc).count() != 0:
    isGestor = True

""" EQUIVALENT SQL """
# SELECT COUNT(teams_player.userInfo_id) FROM teams_player, accounts_users, auth_user
# WHERE teams_player.userInfo_id = accounts_users.cc
# AND accounts_users.user_id = auth_user.id
# AND auth_user.id = userProfile.id
if Player.objects.filter(userInfo_id=userProfile.user_extended.cc).count() != 0:
    isPlayer = True

""" EQUIVALENT SQL """
# SELECT * FROM tournaments_manager, accounts_users, auth_user
# WHERE tournaments_manager.user_id = accounts_users.cc
# AND accounts_users.user_id = auth_user.id
# AND auth_user.id = userProfile.id
# search for all the tournaments the user manages or has managed, returns the manager
tournamentsReference = Manager.objects.filter(user_id=userProfile.user_extended.cc)
```

- Verificar se o utilizador do perfil é manager de algum torneio
- Verificar se o utilizador do perfil é jogador
- Todos os torneios em que o utilizador do perfil é manager


```

# change priv status of player
if 'priv' in request.POST:
    """ EQUIVALENT SQL """
    # UPDATE
    #     accounts_users
    #     INNER JOIN auth_user
    #         ON auth_user.id = accounts_users.user_id
    #     SET
    #         accounts_users.hasPriv = 1
    #     WHERE
    #         auth_user.id = userProfile.id;
    num = Users.objects.filter(user_id=userProfile.id).update(hasPriv=1)
    if num > 0:
        messages.add_message(request, level=messages.INFO, message="User was given privileges successfully!")

# blacklist a user
elif 'blacklist' in request.POST:
    """ EQUIVALENT SQL """
    # UPDATE
    #     teams_player
    #     INNER JOIN accounts_users
    #         ON accounts_users.cc = teams_player.userInfo_id
    #     INNER JOIN auth_user
    #         ON auth_user.id = accounts_users.user_id
    #     SET
    #         teams_player.blacklisted = 1
    #     WHERE
    #         auth_user.id = userProfile.id;
    num = Player.objects.filter(userInfo_id=userProfile.user_extended.cc).update(blacklisted=1)

```

- Alteração de privilégios de um utilizador (ação realizada pelo administrador)

```

if method == "add":
    userBalance += balanceChange
    """ EQUIVALENT SQL """
    # UPDATE
    #     teams_player
    #     INNER JOIN accounts_users
    #         ON accounts_users.cc = teams_player.userInfo_id
    #     INNER JOIN auth_user
    #         ON auth_user.id = accounts_users.user_id
    #     SET
    #         teams_player.balance = userBalance
    #     WHERE
    #         auth_user.id = user.id;
    Player.objects.filter(userInfo=user.user_extended.cc).update(balance=userBalance)
else:
    """ EQUIVALENT SQL """
    # SELECT balance FROM teams_player, accounts_users, auth_user
    # WHERE teams_player.userInfo_id = accounts_users.cc
    # AND accounts_users.user_id = auth_user.id
    # AND auth_user.id = request.user.id
    # check if user has enough balance to remove
    if request.user.user_extended.player.balance < balanceChange:
        messages.error(request, "You don't have THAT much money. No scams allowed.")
        content = {'isGestor': isGestor, 'isPlayer': isPlayer, 'tourReference': tournamentsReference,
                    'userProfile': userProfile, 'balanceForm': bForm}
        return render(request, 'profile.html', content)

```

- Alteração do saldo da conta (ação realizada pelo próprio utilizador)

Ecrã de utilizadores:

```
if filterPeople == "Managers":
    """ EQUIVALENT SQL """
    # select DISTINCT * from auth_user, accounts_users, tournaments_manager
    # WHERE auth_user.id = accounts_users.user_id
    # and tournaments_manager.user_id = accounts_users.cc
    peopleList = User.objects.order_by('first_name').filter(user_extended__manager__isnull=False).distinct()
    title = "Managers"
else:
    filterPeople = request.GET.get('q')
    """ EQUIVALENT SQL """
    # SELECT * from auth_user where is_superuser = False and (auth_user.first_name like '%filterPeople%' or
    # auth_user.last_name like '%filterPeople%')
    peopleList = User.objects.order_by('first_name').filter(is_superuser=False) & (
        User.objects.filter(first_name__icontains=filterPeople) |
        User.objects.filter(last_name__icontains=filterPeople))
    title = "User Search"
```

- Pesquisa por utilizadores que são managers de torneios
- Pesquisa por utilizadores através do seu nome

Editar perfil:

```
phoneNumber = form.cleaned_data['phoneNumber']
bio = form.cleaned_data['bio']
image = request.FILES['image']

folder = 'media/profile-pics'

# save image to the database
fs = FileSystemStorage(location=folder)
filename = fs.save(image.name, image)
file_url = 'profile-pics/' + filename # right url to add to the db

""" EQUIVALENT SQL """
# UPDATE
#   accounts_users
#   INNER JOIN auth_user
#     ON auth_user.id = accounts_users.user_id
#   SET
#     accounts_users.phoneNumber = phoneNumber
#     accounts_users.bio = bio
#     accounts_users.image = file_url
#   WHERE
#     auth_user.id = request.user.id;
Users.objects.filter(cc=request.user.user_extended.cc).update(phoneNumber=phoneNumber, bio=bio,
                                                                image=file_url)
username = request.user.username
```

- Atualizar a informação pessoal e mudança de foto de perfil

NOTA: Não está presente todo o SQL realizado no trabalho, apenas o considerado mais relevante.

--- Detalhes do código relevantes ---

O administrador possui a capacidade de visualizar muito mais do que um utilizador comum, incluindo dar privilégios a outros utilizadores, colocar utilizadores na lista negra. Existe proteção de dados (informações privadas são mantidas privadas, por exemplo o número de cartão de cidadão só pode ser visto pelo próprio utilizador).

--- Extras ---

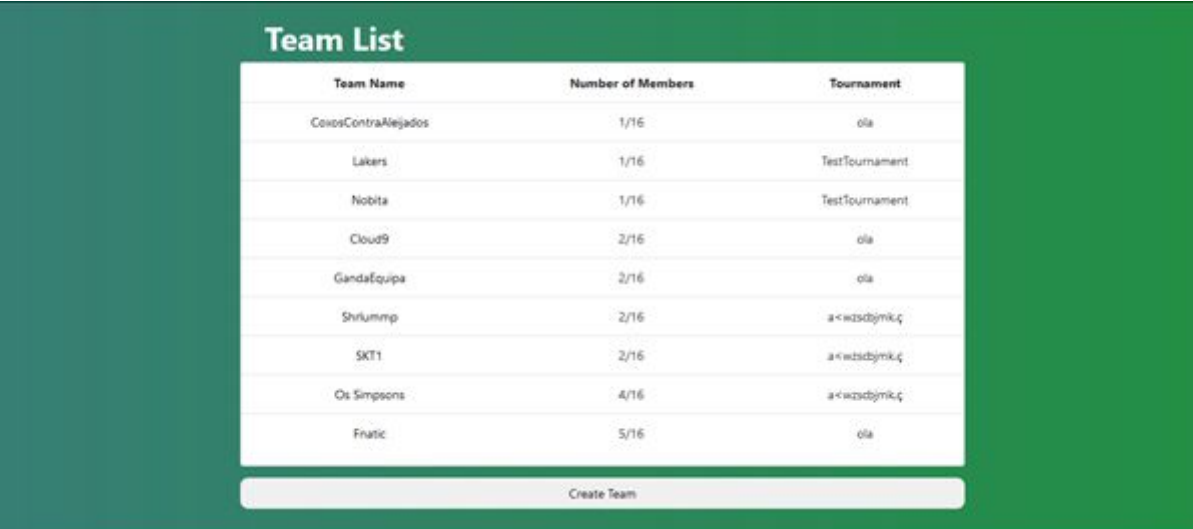
- Utilização de Frameworks: Django e Bootstrap
- Design responsivo da aplicação
- Customização do perfil pessoal (foto de perfil, biografia)
- Base de dados é mantida online: Criei uma máquina na DEICloud onde coloquei a base de dados. Tendo a base de dados online facilitou a visualização dos dados inseridos por todos os membros do grupo. Não só os dados estão disponíveis em tempo real como são evitadas redundância de dados: por exemplo, dois utilizadores que desejam ter o mesmo username não podem ambos registar-se com esse username. Se a base de dados fosse mantida localmente, isso seria possível.

Utilizadores em computadores diferentes conseguem ver todos os utilizadores registados na plataforma, se a base de dados fosse local, apenas conseguiria ver os utilizadores que se registaram usando a sua própria máquina.

Logo, a presença da base de dados num sistema online fez com que o projeto se aproximasse mais de um cenário real, o que era o pretendido.

-- Ecrãs Desenvolvidos --

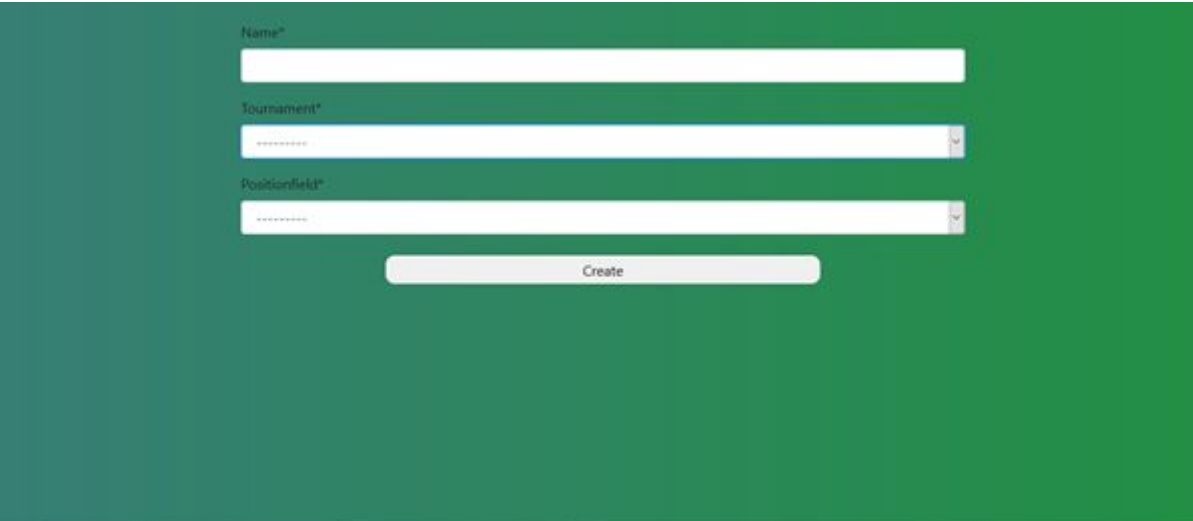
Listagem de Equipas



Team Name	Number of Members	Tournament
CoxosContraAlejados	1/16	ola
Lakers	1/16	TestTournament
Nobita	1/16	TestTournament
Cloud9	2/16	ola
GandaEquipa	2/16	ola
Shrummp	2/16	a<wzsdbynik,ç
SKT1	2/16	a<wzsdbynik,ç
Os Simpsons	4/16	a<wzsdbynik,ç
Fnatic	5/16	ola

Create Team

Criação de uma nova Equipa



Name*

Tournament*

Positionfield*

Create

Detalhes de uma equipa

Nobita

Team Capitan: Test Surname
Total Members: 1
Tournament: TestTournament

Join Team

Team Members

Name	Position
Test Surname	Ataque - Central

Lista onde a pessoa joga

Your Teams List

Active Teams List

Team Name	Number of Members	Tournament
SKT1	2/16	a<wzscbjmkç

Past Teams List

Team Name	Number of Members	Tournament
GandaEquipa	2/16	ola

Área do Capitão

Captain Area

Your Teams

Team Name	Number of Members	Tournament	
GandaEquipa	2/16	ola	<button>Edit</button>

Edição das posições das equipas

GandaEquipa Members

Name/ Username	Old Position	Select Username	Select New Position	
Andre Moreira/ andre	Medio - Central	<input type="text"/>	<input type="text"/>	<button>Save</button>
Marcelo Gomes Gomes/ marcelo	Guarda Redes	<input type="text"/>	<input type="text"/>	<button>Save</button>

-- Código --

Modelos

Todo o código da aplicação web foi desenvolvido usando a framework de Python Django, por tanto a criação das tabelas na base de dados foram feitas no formato de modelos da framework.

Para o registo das equipas, jogadores e suas posições foi criado na base de dados quatro modelos: Player, Team, Position e TeamPlayer que serão registados na base de dados respetivamente como teams_player, teams_team, teams_position e teams_teamplayer.

No modelo Player são registados todos os utilizadores (userInfo) que jogam em alguma equipa, ainda é armazenado o seu saldo (balance) e registado se o jogador está ou não na “lista negra” (blacklisted). Na seguinte imagem está o código do modelo e o seu correspondente em SQL:

```
# CREATE TABLE teams_player (
#   userInfo_id CHAR(255) UNIQUE NOT NULL
#   balance INTEGER NOT NULL DEFAULT 0,
#   blacklisted BOOL NOT NULL DEFAULT false,
#   PRIMARY KEY(userInfo_id)
#   FOREIGN KEY(userInfo_id) REFERENCES accounts_users(user)
# );

class Player(models.Model):
    """
    Player model contains:
    - self user
    - balance
    - isBlackListed
    """

    userInfo = models.OneToOneField(Users, primary_key=True, on_delete=models.CASCADE, unique=True,
                                   related_name="player")

    balance = models.IntegerField(default=0)
    blacklisted = models.BooleanField(default=False)

    def __str__(self):
        return self.userInfo.user.username
```

Um utilizador normal só consegue o estatuto de jogador (Player) quando este se inscreve numa equipa ou cria uma, evitando assim a criação de um jogador sem o registo na plataforma. Os parâmetros balance e blacklisted têm o seu valor inicial como 0 e False, respetivamente.

O método `__str__` é um método do `models.Model`, uma class da framework Django, que está responsável pela forma em como a informação é imprimida quando é feita a pesquisa dela, neste caso foi definido que apareça o username do jogador.

É no modelo Team onde são registadas as equipas criadas contendo o nome (name), o número de membros registados (numbMember), o torneio em que se estão a inscrever (tournament) e o seu capitão de equipa (teamCaptain), que será a pessoa que criou a equipa.

Código SQL:

```
# CREATE TABLE teams_team (  
#     name      VARCHAR(100),  
#     nummembers INTEGER DEFAULT 0,  
#     tournament_id VARCHAR(150) NOT NULL,  
#     PRIMARY KEY(name)  
#     FOREIGN KEY(tournament_id) REFERENCES tournaments_tournaments(name)  
# );
```

Código Python:

```
class Team(models.Model):  
    """  
    Team model contains:  
    - name  
    - number of members  
    - tournament  
    - team Captain  
    """  
  
    name = models.CharField(max_length=100, primary_key=True)  
    numbMembers = models.IntegerField(default=0)  
    tournament = models.ForeignKey(Tournaments, on_delete=models.CASCADE, related_name='tournamentName', default=None)  
    teamCaptain = models.ForeignKey(Player, on_delete=models.CASCADE, related_name="captain", default=None)  
  
    class Meta:  
        unique_together = (('tournament', 'teamCaptain'),)  
  
    def __str__(self):  
        return self.name
```

O nome da equipa tem uma restrição de 100 caracteres e é a chave primária do modelo.

O número de membros da equipa tem um valor inicial 0, apesar de aqui não estar presente qualquer restrição ou valor máximo, não é possível inscrever mais do que 16 elementos na equipa.

Na criação da equipa só é possível seleccionar torneios que existam, o fragmento de código on_delete=model.CASCADE, garante que caso torneio seja apagado a respetiva equipa também seja, evitando informação inválida na base de dados.

O título de capitão de equipa é atribuído ao utilizador ou jogador que crie a equipa.

A classe Meta também pertence à classe models.Model e nesta classe é definido que parâmetros do modelo é que são únicos juntos nomeadamente o 'tournament' e o 'teamCaptain', ou seja, impossibilitando que um jogador crie mais do que uma equipa num torneio.

A tabela `teams_positions` correspondente ao modelo `Position` é estática e não é alterada, mantendo só e unicamente 16 posições disponíveis para escolher: ataque: esquerdo, direito e central, defesa: esquerda, direita e central, médio: esquerdo, direito e central, ponta de lança, guarda redes e três substitutos.

```
# CREATE TABLE teams_position (
#   position VARCHAR(50),
#   PRIMARY KEY(position)
# );
class Position(models.Model):
    """
    Position model contains:
    - position name
    - team where the player is registered
    """

    position = models.CharField(max_length=50, primary_key=True)

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

O modelo `TeamPlayer` é um modelo auxiliar onde será feita a relação entre o jogador, a equipa em que se registou e a sua posição nessa equipa.

```
# CREATE TABLE teamplayer (
#   teamName VARCHAR(100) NOT NULL,
#   playerName VARCHAR(100) NOT NULL,
#   chosenPosition VARCHAR(50) NOT NULL
#   FOREIGN KEY(teamName) REFERENCES teams_team(name)
#   FOREIGN KEY(playerName) REFERENCES teams_player(userInfo_id)
#   FOREIGN KEY(teamName) REFERENCES teams_position(position)
# );
class TeamPlayer(models.Model):
    """
    TeamPlayer model contains:
    - team
    - player
    """

    teamName = models.ForeignKey(Team, on_delete=models.CASCADE, default=None)
    playerName = models.ForeignKey(Player, related_name='teamPlayer', on_delete=models.CASCADE, default=None)
    chosenPosition = models.ForeignKey(Position, related_name='playerPosition', on_delete=models.CASCADE, default=None)

    class Meta:
        unique_together = (('teamName', 'playerName'),)

    def __str__(self):
        return self.teamName.name
```

O parâmetro 'teamName' recebe a equipa em que o jogador se regista, o 'playerName' recebe o jogador e o 'chosenPosition' recebe a posição onde o jogador se inscreveu. Na classe Meta é garantido que um jogador não se pode inscrever numa equipa mais do que uma vez.

Estruturação do código

A framework Django organiza as diferentes aplicações em várias diretorias:

- migrations: onde são registadas as migrações e transformações dos modelos em tabelas para a base de dados.
- templates: onde são guardados os ficheiros HTML
- static: onde são guardados os ficheiros CSS

Na diretoria da aplicação encontram-se os ficheiros responsáveis pela:

- Criação de forms interativos: forms.py
- Criação dos modelos: models.py
- Logica e filtragem das tabelas: views.py
- E a ligação do link à função responsável por mostrar a página: urls.py

views.py

Este ficheiro contém as seguintes funções:

```
class TeamListView(LoginRequiredMixin, ListView):...

@login_required
def showSelfTeamsList(request):...

@login_required
def formTeam(request):...

@login_required
def formJoinTeam(request, name):...

def showNotFullTeams(request):...

# |||||Captain Area|||||

@login_required
def captainArea(request):...

@login_required
def editTeam(request, name):...
```

- TeamListView: são passados para o ficheiro teamsList.html todas as equipas registadas ordenadas pelo número de membros de cada uma.
- showSelfTeamsList: recebe o request como argumento, este request é onde estará registada a informação de que utilizador está a usar a plataforma para poder mostrar ao utilizador em que equipas é que este está registado. As equipas estão também filtradas por equipas registadas em torneios que já passaram:

```
# SELECT
#     teams_team.name,
#     teams_team.tournament_id,
#     tournaments_tournaments.end
# FROM teams_team
# INNER JOIN tournaments_tournaments ON teams_team.tournament_id = tournaments_tournaments.name
# WHERE tournaments_tournaments.end < NOW()
```

e equipas de torneios que estão a decorrer ou que ainda vão ocorrer:


```
# SELECT
#     teams_team.name,
#     teams_team.tournament_id,
#     tournaments_tournaments.end
# FROM teams_team
# INNER JOIN tournaments_tournaments ON teams_team.tournament_id = tournaments_tournaments.name
# WHERE tournaments_tournaments.end >= NOW()
```

- formTeam: responsável por formar o form para criar uma equipa. Inicialmente começa por verificar se o utilizador já tem estatuto de jogador e caso não tenha, atribui-lhe:

```
# SELECT teams_player.userInfo_id, accounts_users.user_id, auth_user.id
# FROM auth_user
# INNER JOIN accounts_users on auth_user.id = accounts_users.user_id
# INNER join teams_player on accounts_users.cc = teams_player.userInfo_id
```

```
# INSERT INTO teams_player (userInfo_id, balance, blacklisted)
# VALUES (accounts_users.cc, 0, False)
```

De seguida, após o registo completo de todos os dados, esta função ainda vai atribuir o estatuto de capitão da equipa ao utilizador e ainda o regista na equipa:

```
# INSERT INTO teams_team (name, numbMembers, tournament_id, teamCaptain_id)
# VALUES (form.fields['name'], 1, form.fields['tournament'], selfPlayer)
```

```
# INSERT INTO teams_teamplayer (playerName_id, teamName_id, chosenPosition_id)
# VALUES (selfPlayer, form, form.positionField)
```

- formJoinTeam: responsável por preparar a informação da equipa a dar display

```
# SELECT * FROM teams_team
# WHERE teams_team.name = 'name'
```

```
# SELECT
#     auth_user. *,
#     teams_player.userInfo_id,
#     accounts_users.user_id,
#     teams_player.blacklisted
# FROM auth_user
# INNER JOIN accounts_users on auth_user.id = accounts_users.user_id
# INNER join teams_player on accounts_users.cc = teams_player.userInfo_id
```

e por registar o utilizador, caso este queira:

```
# INSERT INTO teams_teamplayer (playerName_id, teamName_id, chosenPosition_id)
# VALUES ('request.user.user_extended.player', 'team', 'Position(position)')
```

- showNotFullTeams: filtra as equipas que ainda não estão totalmente preenchidas:

```
# SELECT * from teams_team
# WHERE teams_team.numbMembers < 16
```

- captainArea: esta função está responsável por verificar se o utilizador é capitão de alguma equipa, se é mostra-as e dá a opção de as editar:

```
# SELECT teams_player.userInfo_id, accounts_users.user_id, auth_user.id
# FROM auth_user
# INNER JOIN accounts_users on auth_user.id = accounts_users.user_id
# INNER join teams_player on accounts_users.cc = teams_player.userInfo_id
```

```
# select teams_player.userInfo_id, teams_team.teamCaptain_id
# from teams_team
# JOIN teams_player on teams_team.teamCaptain_id=teams_player.userInfo_id
```

```
# SELECT teams_player.userInfo_id, teams_team.*
# FROM teams_team
# JOIN teams_player on teams_team.teamCaptain_id=teams_player.userInfo_id
# WHERE teams_teamCaptain_id = 'request.user.user_extended.player'
```

- editTeam: esta função está responsável por guardar as alterações que o capitão fez:

```
# UPDATE teams_teamplayer
# SET teams_teamplayer.chosenPosition_id = 'position'
# WHERE teams_teamplayer.teamName_id = 'team' AND teams_teamplayer.playerName = 'player'
```

Nota: Não está aqui disposto todo o SQL, apenas o fundamental para as funções

-- Extras --

- Utilização das frameworks: Django e Bootstrap
- Design responsivo

Ricardo Figueiredo

--- Ecrãs desenvolvidos ---

Lista de Torneios:

Tournament Manager

Home

Tournaments

People ▾

My Profile

Logout

Name:

Order by: Name ▾

Order: Ascending ▾

Show only: Show All ▾

Search

Name	Start Date	End Date	Number Of Teams
a<wzscbjmkç	Feb. 1, 2020	Feb. 2, 2020	3
ola	Nov. 8, 2019	Nov. 29, 2019	4
raquelTour	Dec. 1, 2019	Dec. 20, 2019	0
TestTournament	Dec. 1, 2019	Dec. 31, 2019	2
Tournament	Dec. 1, 2019	Dec. 31, 2019	0

New Tournament

Detalhes de um Torneio:

Tournament Manager				Home	Tournaments	People ▾	My Profile	Logout
Name:		Example						
Start Date:		Nov. 1, 2019						
End Date:		Nov. 31, 2019						
Manager Name:		Example Manager						
Days of the week:		Monday Wednesday Friday						
Teams:								
Team Name: Captain:		Number of Players:						
Lakers		Rui Matos		1				
Nobita		Test Surname 1						
Games:								
Teams		Day	Field	Results				
Team1 vs Team2		Nov. 2, 2019	Arca D'Água	2-2				
Team3 vs Team4		Nov. 11, 2019	Estádio da UC	4-0				
Number of games to Generate:		<input type="text" value="1"/>		Generate Games				

Tournament Manager				Home	Tournaments	People ▾	My Profile	Logout
Name:		Tournament						
Start Date:		Dec. 1, 2019						
End Date:		Dec. 31, 2019						
Manager Name:		Test Surname						
Days of the week:		Monday Wednesday Friday						
Teams:								
		No teams added yet						
Games:								
		No games created yet						
Number of games to Generate:		<input type="text" value="1"/>		Generate Games				

Páginas de criação de um torneio:

Tournament Manager

HomeTournamentsPeopleMy Profile

Logout

Name:

Start:

dd/mm/yyyy

End:

dd/mm/yyyy

Monday: ☐

Tuesday: ☐

Wednesday: ☐

Thursday: ☐

Friday: ☐

Saturday: ☐

Sunday: ☐

Next Step

Forbidden Dates:

Date:

2019-12-02

IsForbidden:

☒

Date:

2019-12-04

IsForbidden:

☒

Date:

2019-12-06

IsForbidden:

☐

Date:

2019-12-09

IsForbidden:

☐

Date:

2019-12-11

IsForbidden:

☐

Date:

2019-12-13

IsForbidden:

☐

Date:

2019-12-16

IsForbidden:

☐

Date:

2019-12-18

IsForbidden:

☐

Date:

2019-12-20

IsForbidden:

☐

Date:

2019-12-23

IsForbidden:

☒

Date:

2019-12-25

IsForbidden:

☐

Date:

2019-12-27

IsForbidden:

☐

Date:

2019-12-30

IsForbidden:

☐

Submit

Criar jogos:

Team A:

GandaEquipa

Team B:

Cloud9

Slot:

December 13 2019

Field:

Estádio da UC

Team A:

Fnatic

Team B:

CoxosContraAleijados

Slot:

December 18 2019

Field:

Campo Hospital dos Covões

Team A:

Cloud9

Team B:

Fnatic

Slot:

December 23 2019

Field:

Arca D'Água

Submit

Team A: <input type="text" value="CoxosContraAleijados"/>	Team B: <input type="text" value="Cloud9"/>	Slot: <input type="text" value="December 16 2019"/>	Field: <input type="text" value="Associação Académica de Coimbra"/>
Team A: <input type="text" value="Cloud9"/>	Team B: <input type="text" value="Fnatic"/>	Slot: <input type="text" value="December 13 2019"/>	Field: <input type="text" value="Campo de Santa Cruz"/>
Team A: <input type="text" value="GandaEquipa"/>	Team B: <input type="text" value="CoxosContraAleijados"/>	Slot: <input type="text" value="December 23 2019"/>	Field: <input type="text" value="Estádio da UP"/>
Team A: <input type="text" value="Fnatic"/>	Team B: <input type="text" value="GandaEquipa"/>	Slot: <input type="text" value="December 18 2019"/>	Field: <input type="text" value="Estádio da UC"/>

--- Entidades---

A classe Tournament e respetiva entidade tournaments_tournament representam um torneio registado no sistema, o código em python e respetivo sql apresentam-se abaixo:

```
class Tournaments(models.Model):
    name = models.CharField(max_length=150, primary_key=True)
    start = models.DateField(null=False)
    end = models.DateField(null=False)
    monday = models.BooleanField(null=False, default=False)
    tuesday = models.BooleanField(null=False, default=False)
    wednesday = models.BooleanField(null=False, default=False)
    thursday = models.BooleanField(null=False, default=False)
    friday = models.BooleanField(null=False, default=False)
    saturday = models.BooleanField(null=False, default=False)
    sunday = models.BooleanField(null=False, default=False)

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

```
# CREATE TABLE tournaments_tournaments (
#   name      varchar(150),
#   start date NOT NULL,
#   end   date NOT NULL,
#   friday  boolean NOT NULL,
#   monday  boolean NOT NULL,
#   saturday boolean NOT NULL,
#   sunday  boolean NOT NULL,
#   thursday boolean NOT NULL,
#   tuesday  boolean NOT NULL,
#   wednesday boolean NOT NULL,
#   PRIMARY KEY(name)
# );
```

Esta entidade possui as seguintes colunas:

- name: nome do torneio e seu identificador único (primary key), limite máximo 150 caracteres
- start, end: data de início e fim do torneio, respetivamente.
- monday, tuesday, wednesday, thursday, friday, saturday, sunday: conjunto de booleans que permitem guardar e posteriormente visualizar em que dias da semana o torneio acontece

No momento de criação do torneio as seguintes validações são aplicadas (através da classe form da framework Django):

```
if start and start < datetime.now().date():
    self.add_error('start', "The tournament can't have started already")

if end and end < datetime.now().date():
    self.add_error('end', "The tournament can't have ended already")

if start and end and end < start:
    self.add_error('end', "The tournament can't end before it starts")

if monday + tuesday + wednesday + thursday + friday + saturday + sunday == 0:
    self.add_error('monday', "Select atleast one day of the Week")
```

- (não presente na imagem) Todos os campos têm de estar preenchidos
- Não é permitida a criação de um torneio que já começou
- Não é permitida a criação de um torneio que já acabou
- Não é permitida a criação de um torneio que acaba antes de começar
- É necessário escolher pelo menos um dia da semana para o torneio acontecer

A entidade manager permite saber quem é o gerente de um determinado torneio

```
class Manager(models.Model):
    user = models.ForeignKey('accounts.Users', on_delete=models.CASCADE,
                             null=False, related_name="manager")
    tournament = models.ForeignKey(Tournaments, on_delete=models.CASCADE,
                                   null=False, related_name="manager_table")

    class Meta:
        unique_together = (('user', 'tournament'),) # composite key
```



```
# CREATE TABLE tournaments_manager (
#   id          int,
#   user_id     bigint NOT NULL,
#   tournament_id varchar(150) NOT NULL,
#   PRIMARY KEY(id)
# );
```

```
# ALTER TABLE manager ADD CONSTRAINT manager_fk1 FOREIGN KEY (user_id) REFERENCES users(cc);
# ALTER TABLE manager ADD CONSTRAINT manager_fk2 FOREIGN KEY (tournament_id) REFERENCES tournaments(name);
```

Esta entidade possui os seguintes campos:

- id: identificador único
- user_id: referência ao utilizador gerente (foreign key)
- tournament_id: referência ao torneio gerido pelo utilizador (foreign key)

A esta entidade são aplicadas as seguintes validações:

- Nenhuma coluna pode estar vazia
- Nenhuma linha pode ter um conjunto igual de chaves estrangeiras

A entidade TournamentSlots serve para guardar os dias em que há jogos, e também os dias em que deveria haver mas excepcionalmente não há:

```
class TournamentSlots(models.Model):
    date = models.DateField(null=False)
    tournament = models.ForeignKey(Tournaments, on_delete=models.CASCADE,
                                   null=False, related_name='slots')
    isForbidden = models.BooleanField(null=False, default=False)

    class Meta:
        unique_together = (('tournament', 'date'),)

    def __str__(self):
        return self.date.strftime("%B %d %Y")
```

```
#
# CREATE TABLE tournaments_tournamentslots (
#   id          int,
#   isforbidden boolean NOT NULL,
#   date        date NOT NULL,
#   tournament_id varchar(150) NOT NULL,
#   PRIMARY KEY(id)
# );
```

```
ALTER TABLE tournamentslots ADD CONSTRAINT tournamentslots_fk1 FOREIGN KEY (tournament_id) REFERENCES tournaments(name);
```

A esta classe são aplicadas as seguintes validações:

- Todos os campos têm de ser preenchidos
- As combinações de torneio e data são únicas

A entidade GameFields guarda as informações dos campos de futebol registados no sistema

```
class GameFields(models.Model):
    name = models.CharField(max_length=150, primary_key=True)
    local = models.CharField(max_length=150, null=False)
    price = models.IntegerField(null=False, validators=[MinValueValidator(0)])

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

```
# CREATE TABLE tournaments_gamefields (
#   name      varchar(150),
#   local     varchar(512) NOT NULL,
#   price     int NOT NULL,
#   PRIMARY KEY(name)
# );
```

Os parâmetros são:

- name: nome do campo, identificador único
- local: local onde o campo se situa
- price: preço de aluguer do campo

As validações aplicadas a esta entidade são:

- Nenhum campo vazio
- Campo *price* tem de ser maior ou igual a 0

A entidade Games guarda a informação dos jogos registados na base de dados:

```
class Games(models.Model):
    teamA = models.ForeignKey('teams.Team', null=False, on_delete=models.CASCADE,
                              related_name='teamA')
    teamB = models.ForeignKey('teams.Team', null=False, on_delete=models.CASCADE,
                              related_name='teamB')
    slot = models.ForeignKey(TournamentSlots, null=False, on_delete=models.CASCADE,
                             related_name='game')
    field = models.ForeignKey(GameFields, null=False,
                              on_delete=models.CASCADE)

    class Meta:
        unique_together = (('slot', 'field'), ('teamA', 'teamB', 'slot'),)
```

```
# CREATE TABLE tournaments_game (
#   id          int,
#   slot_id int NOT NULL,
#   teamA       varchar(100) NOT NULL,
#   teamB       varchar(100) NOT NULL,
#   field_id    varchar(150) NOT NULL,
#   PRIMARY KEY(id)
# );
```

```
# ALTER TABLE game ADD CONSTRAINT game_fk1 FOREIGN KEY (slot_id) REFERENCES tournamentslots(id);
# ALTER TABLE game ADD CONSTRAINT game_fk2 FOREIGN KEY (teamA) REFERENCES team(name);
# ALTER TABLE game ADD CONSTRAINT game_fk3 FOREIGN KEY (teamB) REFERENCES team(name);
# ALTER TABLE game ADD CONSTRAINT game_fk4 FOREIGN KEY (field_id) REFERENCES gamefields(name);
```

Esta entidade apresenta as seguintes colunas:

- id: identificador único, número sequencial
- teamA e teamB: referências às duas equipas que se vão defrontar
- field_id: referencia ao campo onde o jogo decorrerá
- slot_id: referência à data em que decorrerá o jogo

As validações aplicadas a esta entidade são:

- A combinação slot_id e field_id tem de ser única, ou seja não podem ocorrer mais do que um jogo num campo de cada vez
- a combinação de teamA, teamB e slot_id tem de ser única, para evitar a possível duplicação de um registo
- (No momento da criação de um jogo) Uma equipa não pode defrontar-se

A entidade PlayerGoals permite visualizar facilmente quantos golos um determinado jogador marcou, e em que jogo

```

class PlayerGoals(models.Model):
    player = models.ForeignKey('teams.Player', null=False,
                              on_delete=models.CASCADE,
                              related_name='playerGoalsplayer')
    goals = models.IntegerField(null=False,
                               validators=[MinValueValidator(1)])
    game = models.ForeignKey(Games, null=False,
                            on_delete=models.CASCADE, related_name='playerGoalsgame')

    class Meta:
        unique_together = (('game', 'player'),)

```

```

# CREATE TABLE tournaments_playergoals (
#   id          int,
#   goals       int,
#   player_id   bigint NOT NULL,
#   game_id     int NOT NULL,
#   PRIMARY KEY(id)
# );

```

```

ALTER TABLE playergoals ADD CONSTRAINT playergoals_fk1 FOREIGN KEY (player_id) REFERENCES
player(user_id);

```

```

ALTER TABLE playergoals ADD CONSTRAINT playergoals_fk2 FOREIGN KEY (game_id) REFERENCES
game(id);

```

Esta entidade possui os seguintes campos:

- id: inteiro, identificador único do registo
- goals: número de golos marcados
- player_id: referência ao jogador
- game_id: referência ao jogo em que os golos foram marcados

Nesta entidade são aplicadas as seguintes validações:

- Não são permitidos campos vazios
- Os golos de um jogador num determinado jogo só podem ser registados uma vez

--- Estruturação do código ---

A framework Django permite dividir o projeto em várias *apps* o que permitiu uma melhor divisão do trabalho, pois apesar do código estar separado, pode ser facilmente referenciado. A minha parte deste projeto encontra-se na app 'tournaments'.

Na diretoria tournaments encontram-se várias subdiretorias: migrations, static e templates.

A pasta migrations possui os ficheiros de migração, que permitem a framework guardar e aplicar alterações feitas aos modelos, evitando assim a reconstrução da base de dados. A pasta static possui

os ficheiros css utilizados. Finalmente, na pasta templates encontram-se os ficheiros html utilizados neste projeto.

Ainda dentro da pasta tournaments podem ser encontrados os seguintes ficheiros (apenas serão referidos os ficheiros a serem notados):

- models.py
- forms.py
- urls.py
- views.py

O ficheiro models.py possui os modelos utilizados nesta app, já listados anteriormente. No ficheiro forms.py encontram-se os formulários utilizados para inserção de inputs por parte do utilizador. O ficheiro urls.py contém os padrões de urls que o Django irá analisar ao navegar pela aplicação (que urls existem, que parâmetros aceitam, qual a sua view, e qual o seu nome para fácil referência). O ficheiro views.py possui a lógica de cada página, sendo invocada sempre que um pedido HTML é feito.

As views incluídas neste último ficheiro são:

```
@login_required
def tournamentListView(request):...

def tournamentDetails(request, tournament_name):...

@login_required
def newTournamentFormView(request):...

@login_required
def forbiddenDatesView(request, tournament_name):...

@login_required()
def createGamesView(request, tournament_name, numGames):...
```

- tournamentListView: ao receber um pedido HTML apresenta a tabela de torneios, aplicando os filtros e ordenamentos especificados pelo utilizador
- tournamentDetails: ao receber além do pedido o nome de um torneio, apresenta na página detalhes sobre esse torneio
- newTournamentFormView: apresenta o formulário de criação de um torneio e processa esse mesmo formulário aquando da sua submissão
- forbiddenDatesView: apresenta o formulário de escolha de dias em que não poderá haver jogo e processa o input

- createGamesView: mostra o formulário de criação de jogos e processa os inputs do utilizador

--- SQL relevante em cada ecrã ---

Lista de torneios:

```
def tournamentListView(request):
    # SELECT tournaments_tournaments.name, tournaments_tournaments.start,
    # tournaments_tournaments.end, count(teams_team.name) AS 'ntteams' FROM tournaments_tournaments
    # LEFT OUTER JOIN teams_team ON teams_team.tournament_id = tournaments_tournaments.name
    # GROUP BY tournaments_tournaments.name
    query = Tournaments.objects.annotate(ntteams=Count('tournamentName')).values('name', 'end', 'start', 'ntteams')

    namefilter = request.GET.get('name')
    if namefilter:
        # SELECT ...
        # WHERE tournaments_tournaments.name LIKE '%namefilter%'
        # GROUP BY...
        query = query.filter(name__icontains=request.GET.get('name'))

    # Nas linhas seguintes: substituir 'WHERE' por 'AND' caso o filtro anterior seja aplicado
    if request.GET.get('show_only') == 'c':
        # WHERE tournaments_tournaments.end < Now()
        query = query.filter(end__lt=datetime.datetime.now().date())
    elif request.GET.get('show_only') == 'f':
        # WHERE tournaments_tournaments.start > Now()
        query = query.filter(start__gt=datetime.datetime.now().date())
    elif request.GET.get('show_only') == 'h':
        # WHERE Now() BETWEEN tournaments_tournaments.start and tournaments_tournaments.end
```

Nesta página é explorado o conceito “QuerySets are Lazy” (retirado da documentação oficial da framework), o que permite que sejam aplicados filtros sucessivos a uma consulta sem haver acessos à base de dados, permitindo assim um código mais legível e otimizado. De notar que o SQL comentado no código desta view foi sumariado com reticências para evitar repetições.

Conceitos explorados nesta página:

- Pesquisa filtrada
- Ordenamento de tabelas
- Outer Join
- Criação dinâmica de colunas

Detalhes de torneio:

```

def tournamentDetails(request, tournament_name):
    if request.method == 'GET':
        # SELECT tournaments_tournaments.*,CONCAT(auth_user.first_name, ' ',auth_user.last_name) as manager
        # FROM tournaments_tournaments
        # JOIN tournaments_manager ON tournaments_manager.tournament_id = tournaments_tournaments.name
        # JOIN accounts_users ON tournaments_manager.user_id = accounts_users.cc
        # JOIN auth_user ON accounts_users.user_id = auth_user.id
        # WHERE tournaments_tournaments.name = tournament_name
        t = Tournaments.objects.filter(name=tournament_name).annotate(
            manager=Concat(F('manager_table_user_user_first_name'), Value(' '),
                           F('manager_table_user_user_last_name')).first()

        # SELECT name, CONCAT(first_name, ' ',last_name), numbMembers as capname FROM teams_team
        # JOIN teams_player ON teams_team.teamCaptain_id = teams_player.userInfo_id
        # JOIN accounts_users ON teams_player.userInfo_id = accounts_users.cc
        # JOIN auth_user ON accounts_users.user_id = auth_user.id
        # WHERE tournament_id = t.name
        teams = Team.objects.filter(tournament_id=t.name).annotate(
            capname=Concat(F('teamCaptain_userInfo_user_first_name'), Value(' '),
                           F('teamCaptain_userInfo_user_last_name')).values('name', 'capname', 'numbMembers')

        # SELECT CONCAT(teamA_id,' vs ', teamB_id) teams, tournaments_tournamentslots.date, field_id,
        # COALESCE(Sum((SELECT goals FROM tournaments_playergoals pg
        # JOIN teams_player p ON pg.player_id = p.userInfo_id
        # JOIN teams_teamplayer tp ON tp.playerName_id = p.userInfo_id
        # WHERE game_id = game.id AND tp.teamName_id = game.teamA_id)),0) goalsA,
        # COALESCE(Sum((SELECT goals FROM tournaments_playergoals pg
        # JOIN teams_player p ON pg.player_id = p.userInfo_id

```

Nesta página podem ser vistas consultas dotadas de uma especial complexidade, sendo necessárias agregações de várias tabelas de forma e de subconsultas a obter dados, assim tirando proveito das capacidades da linguagem SQL de forma a evitar redundância de dados.

Conceitos explorados nesta página:

- Concatenação de strings
- Junção de tabelas
- Subconsultas
- Agrupamentos
- Somatórios
- Inner Join

Apesar de também ser usado SQL nas restantes páginas, estes dois exemplos são de salientar pela complexidade e exploração das capacidades tanto da framework como da base de dados, sendo as restantes utilizações simples inserções e atualizações sem nenhuma característica a realçar.

--- Extras ---

- Utilização de Frameworks: Django e Bootstrap
- Design Responsivo da aplicação
- Cálculo automático dos dias em que poderá haver jogos