SoccerNow

Luis Santos fc58437
Liliana Valente fc59846
Denis Bahnari fc59878

no âmbito da disciplina
Construção de Sistemas de Software

Ciências ULisboa

Contents

Diagrama de Classes

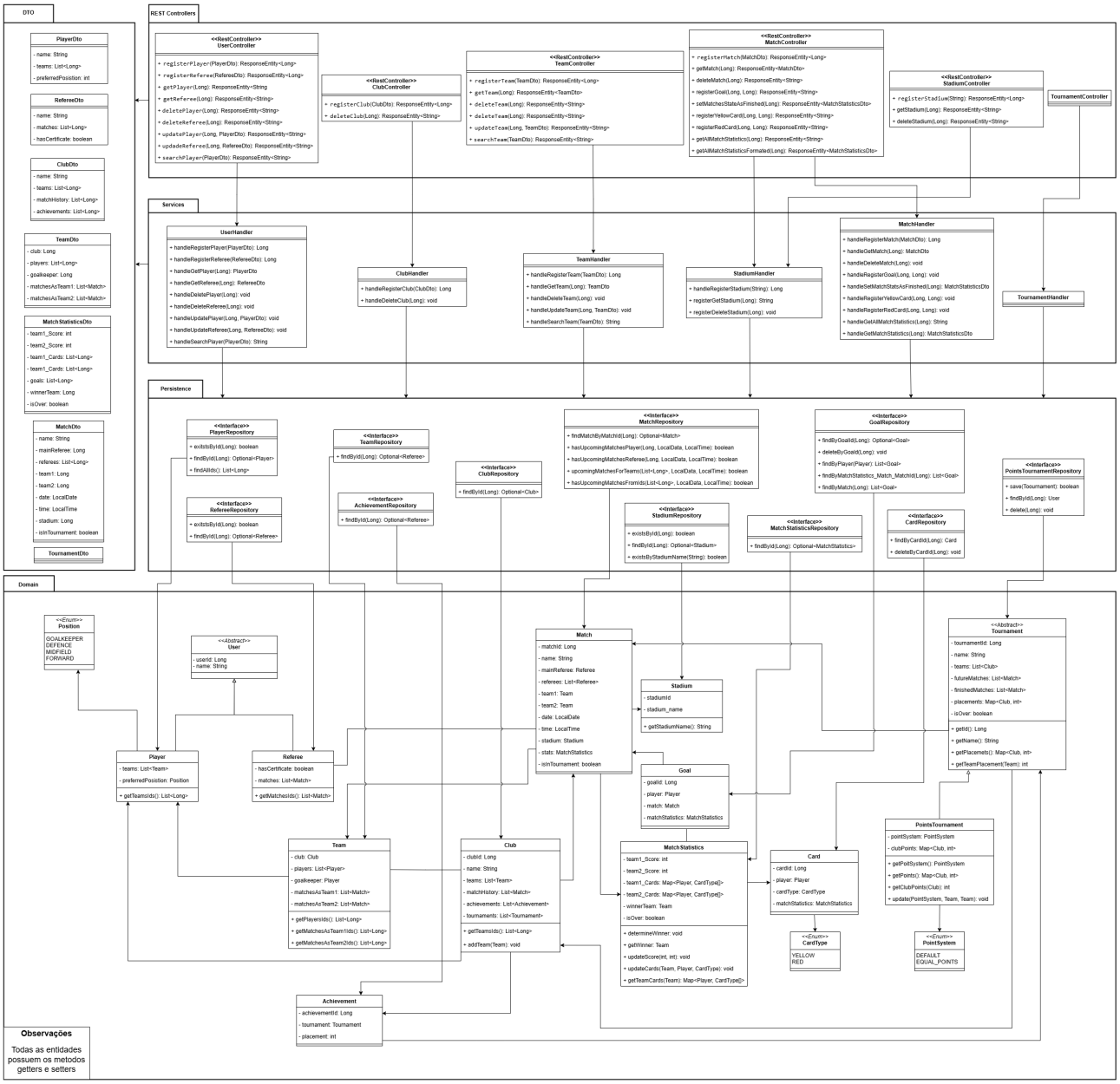
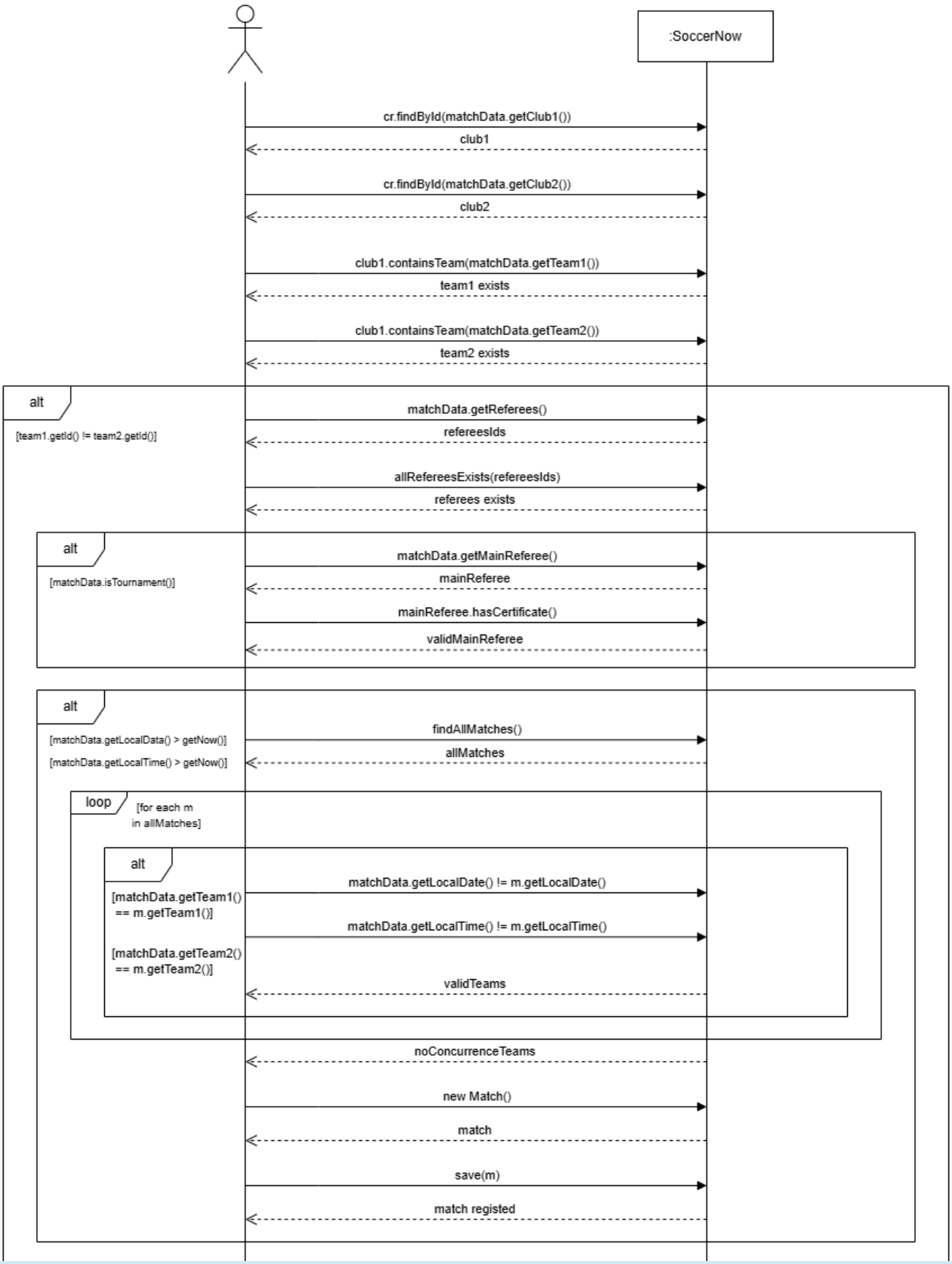


Diagrama de Sequências de Sistema

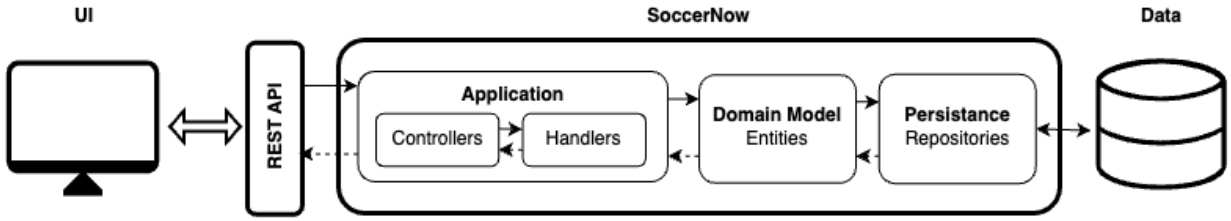


Estrutura e divisão por camadas

Estruturamos o código seguindo uma arquitetura em camadas que separa claramente as responsabilidades da aplicação SoccerNow.

A camada de controller lida com os pedidos HTTP que são acedidos atravé de REST Endpoints no Swagger e atua como ponto de entrada para o sistema, encaminhando as requisições para os handlers apropriados. A camada de service, que contém os handlers invocados pelos controllers, contém também toda a lógica de negócio da aplicação, centralizando todas as transações para os repositórios e camada de dados. A camada de repository comunica diretamente com a base de dados, utilizando o mapeamento ORM com JPA anotations para acesso e persistência de dados na BD.

Acreditamos que esta divisão clara em camadas promove a escalabilidade e manutenção do sistema.



Foram ainda realizados, ao longo do desenvolvimento da aplicação, testes que numa primeira fase visavam testar a eficácia do mapeamento JPA e posteriormente passaram a testar os endpoints REST.

Visão Geral do Funcionamento da Aplicação

A aplicação SoccerNow foi construida para seguir um determinado fluxo e deve ser testada da seguinte forma:

- Começar por criar árbitros e jogadores
- Criar um clube
- Criar uma equipa e a associar os respetivos jogadores e clube
- Criar um estádio
- Criar um jogo

Abaixo encontram-se ainda representados todos os enpoints com explemplos de como podem ser acedidos.

Player Endpoints

POST `/users/register/player`

```
"name": "player1",
"teams": [1, 2, 5],
"preferedPosition": "DEFENDER"
```

Para registar um jogador, é preciso usar o endpoint POST, que requer o corpo JSON com o do nome do jogador, as equipas a que o jogador pertençe e a posição preferida do jogador.
O campo "name" não pode ser null nem vazio, "teams" apenas aceita registar equipas válidas que existem na aplicação, sendo permitido enviar lista vazia para registar jogador sem equipas, e "preferedPosition" aceita apenas 4 posições válidas: GOALKEEPER, DEFENDER, MIDFIELDER e FORWARD.

GET `/users/player/{id}`

```
"name": "player1",
"teams": [1, 2, 5],
"preferedPosition": "DEFENDER"
```

Para obter as informações de um jogador, basta usar o endpoint GET, que requer apenas o ID do jogador que se pretende obter. Este endpoint devolve um JSON com todas os atributos do jogador.

DELETE `/users/player/{id}`

```
Player with ID 1 removed!
```

Para remover um jogador, usa-se o endpoint DELETE, requer o ID do jogador que se pretende remover.

PUT `/users/player/{id}`

```
"name": "PlayerNameUpdated",
"teams": [],
"preferedPosition": "GOALKEEPER"
```

```
Player with ID 1 updated!
```

Para atualizar um jogador, usa-se o endpoint PUT, que requer o ID do jogador a ser atualizado, assim como o corpo JSON com as novas informações do jogador, que vão ser substituidas.

GET /users/player/search

```
"name": "PlayerWithThisName",
"preferedPosition": "GOALKEEPER"
```

Para pesquisar jogadores, usa-se o endpoint GET, que requer o corpo JSON com os atributos que se pretente obter o jogador, cada campo no JSON vai filtrar a pesquisa dos jogadore, nenhum campo é obrigatório.

Referee Endpoints

POST /users/register/referee

```
"name": "RefereeName",
"hasCertificate": true,
"matches": [1, 2, 3]
```

Para adicionar um árbito, é preciso usar o endpoint POST, com o corpo JSON de todos os atributos do àrbito. O campo "name" com o nome do àrbito não pode ser null nem vazio, "hasCertificate" determina se o àrbito possui ou não certificado, e "matches" representa todas as partidas que o àrbito tem para arbitrar, que existam na aplicação e que possuam data/hora no futuro, podendo ser passada uma lista vazia quando não existem jogos.

GET /users/referee/{id}

```
"name": "Referee1",
"hasCertificate": true,
"matches": []
```

Para obter as informações sobre um àrbito, usa-se o endpoint GET, com o ID do àrbito, devolvendo o JSON com todos os atributos do àrbito.

DELETE /users/referee/{id}

```
Referee with ID 2 removed!
```

Para remover um àrbito, usa-se o endpoint DELETE, requer o ID do àrbito que se pretende remover da aplicação.

PUT /users/referee/{id}

```
"name": "referee1Updated",
"hasCertificate": false,
"matches": [1]
```

```
Referee with ID 2 updated!
```

Para atualizar um àrbito, usa-se o endpoint PUT, que requer o ID do àrbito a ser atualizado, assim como o corpo JSON com as novas informaçoes do àrbito, que vão ser substituidas.

Club Endpoints

POST `/register/club`

```
"name": "Sporting FC",
"tournament": []
```

Para registar um clube na aplicação, é preciso usar o endpoint POST com informação sobre o clube, em formato JSON. No caso "name" é o nome do clube, que não pode ser nem null nem vazio, "tournament" representa os torneios em que o clube atualmente particia.
Para o registo de clubes decidiu-se não registar inicialmente equipas, pois todas as equipas da aplicação já têm um clube associado, pelo que nunca ia existir equipas disponiveis para adicionar ao novo clube. Também não se pede o historico de partidas, que é registado sempre automáticamente quando um clube vence/perde um jogo, o mesmo se aplicando para os achievements do clube.

GET `/club/matchHistory/{id}`

```
MatchHistory: [1, 2, 7, 10]
```

Para se obter o histórico de partidas de um clube, usa-se o endpoint GET, que requer o ID do clube, devolvendo o ID de todas as partidas onde uma equipa do clube já participou.

DELETE `/club/{id}`

```
Club with ID 1 removed!
```

Para remover um clube da aplicação, é usado o endpoint DELETE, que usa o ID do clube pretendido, e remove tanto o clube, como todas as suas equipas assiciadas da aplicação.

Stadium Endpoints

POST `/register/stadium/{stadiumName}`

Para registar um estádio na aplicação, basta usar o endpoint POST, com o nome pretendido no path.

GET `/stadium/{stadiumId}`

Para obter os dados do estádio, usa-se o endpoint GET com o ID do estádio pretendido.

DELETE `/stadium/{stadiumId}`

Para remover um estádio, basta usar o endpoint REMOVE, com o ID do estádio.

Team Endpoints

POST /register/team

```
"club": 1,
"players": [1, 5],
"goalkeeper": 5,
"matchesAsTeam1": [1],
"matchesAsTeam2": []
```

Para registar uma equipa, é necessário usar o endpoint POST, com o corpo JSON dos atributos da team. O "club" guarda o id do clube da equipa, "palyers" são a lista dos jogadores da equipa (pode ser vazia), "goalkeeper" é o ID do quarda-redes da equipa, e "matchesAsTeam1" e "matchesAsTeam2" são os futuros jogos que a equipa tem por realizar.

GET /team/{id}

```
"club": 1,
"players": [1, 5],
"goalkeeper": 4,
"matchesAsTeam1": [],
"matchesAsTeam2": []
```

Para obter os dados de uma equipa, basta usar o endpoint GET com o ID da equipa, devolvendo o JSON com todos os atributos da equipa.

DELETE /team/{id}

```
Team with ID 1 removed!
```

Para remover uma equipa da aplicação, usa-se o endpoint DELETE, com o ID da equipa, removendo a equipa da base de dados.

PUT /team/{id}

```
"club": 2,
"players": [],
"goalkeeper": 4,
"matchesAsTeam1": [],
"matchesAsTeam2": [1, 2, 6]
```

```
Match with Id 1 does not exist!
```

Para atualizar uma equipa, usa-se o endpoint PUT, que requer o ID da equipa, assim como o corpo JSON com os novos atributos da equipa para substituir.

GET /team/search

```
"club": 1,
"goalkeeper": 4,
"matchesAsTeam2": []
```

```
Teams: [1]
```

Para pesquisar equipas, usa-se o endpoint GET, que requer o corpo JSON com os atributos que se pretente obter a equipa, cada campo no JSON vai filtrar a pesquisa das equipas, não sendo nenhum dos campos obrigatório.

Match Endpoints

POST /register/match

```
"principalReferee": 4,
"referees": [4, 5],
"team1": 1,
"team2": 2,
"date": "2025-05-04",
"time": "12:00:00",
"stadium": 1,
"stats": 0,
"tournament": 0
```

ou entao caso só haja 1 árbitro na partida

```
"principalReferee": 4,
"referees": [4],
"team1": 1,
"team2": 2,
"date": "2025-05-04",
"time": "12:00:00",
"stadium": 1,
"stats": 0,
"tournament": 0
```

Para registar um jogo na aplicação, é necessário usar o endpoint POST, que requer o copor JSON com todas as informações sobre o jogo a ser realizado. O corpo requer o ”principalReferee” que precisa de estar presente entre todos os ”referees”, a ”team1” e ”team2” são as equipas que vão jogar o jogo, precisa de ”date” e ”time”, que têm de ser no futuro, assim como o ID do ”stadium” onde vai ser realizado o jogo, os ”status” caso se pretenda já relacionar algum existende (na abstenção é criado automáticamente), e por fim ”tournament” que ainda não foi implementado. O sistema só permite que seja registados jogos que irão acontecer no futuro.

NOTA: devido a um problema de formatação do LocalTime no Swagger(que nao conseguimos resolver). É necessário alterar o campo ”time” para ficar igual ao exemplo acima.[1]

[1]Referência aqui: https://github.com/quarkusio/quarkus/issues/6065

GET `/match/{id}`


```
"principalReferee": 4,
  "referees": [4, 5],
  "team1": 1,
  "team2": 2,
  "date": "2025-05-04",
  "time": "12:00:00",
  "stadium": 1,
  "stats": 1,
  "tournament": null
```


Para obter os dados de um jogo, basta usar o endpoint GET com o ID do match, devolvendo o JSON com todos os atributos do match.

GET `/match/{matchId}/getStatistics`


```
  "matchStatisticsId": 1,
  "team1_Score": 0,
  "team2_Score": 1,
  "team1_Cards": [],
  "team2_Cards": [
    4,
    5
  ],
  "goals": [
    2
  ],
  "winnerTeam": 2,
  "over": true
```


Para obter as estatisticas de um jogo na aplicação usamos este endpoint GET com o ID do match.

GET `/match/{matchId}/getAllStatisticsFormated`

```
Match ID: 3
Match Date: 2025-05-03
Match Time: 18:00
Stadium: Est dio das Portas do Sol
Team 1: Equipa N o Fixe
Team 2: Equipa Fixe
Winner Team: Equipa Fixe
Team 1 Score: 0
Team 2 Score: 1
Red Cards:
Denis
Lili
Yellow Cards:
Goals:
Lili
```

Este endpoint nada mais faz do que mostrar as estatísticas de forma visual aos visualizadores dos endpoints. Na próxima iteração será removido, serve portanto somente o propósito de visualização clara das informações que estão a ser acedidas. No entanto, detetamos que apresenta um bug pois duplica cartões, no entanto não é muito relevante pois o getStatistics apresenta a informação correta e este endpoint será removido na próxima fase.

DELETE `/match/{id}`

Para remover um match da aplicação, usa-se o endpoint DELETE, com o ID da equipa, removendo a equipa da base de dados. A equipa só pode ser removida se o jogo ainda não tiver sido jogado. Consideramos que não devemos prmitir que jogos que já foram jogados sejam excluidos da base de dados.[2]

PUT `/match/{matchId}/register/goal/player/{playerId}`

Para adicionar um golo a um determinado jogador num jogo é necessário usar o endpoint PUT que requer o ID do match e o ID do jogador

PUT `/match/{matchId}/register/player/{playerId}/card/yellowCard`

Para adicionar um cartão amarelo a um determinado jogador num jogo é necessário usar o endpoint PUT que requer o ID do match e o ID do jogador.

[2]Consideramos que jogos não jogados são jogos nos quais não tenham sido atribuidos nem jogos nem cartoes

PUT `/match/{matchId}/register/player/{playerId}/card/redCard`

Para adicionar um cartão vermelho a um determinado jogador num jogo é necessário usar o endpoint PUT que requer o ID do match e o ID do jogador.

PUT `/match/{matchId}/setStateAsFinished`

Para mudar o estado de um jogo para terminado é necessário usar o endpoint PUT que requer o ID do match. Este endpoint retorna a mensagem:

```
"Match state is set to finished"
```

Mapeamento JPA

Mapeamento da entidade Achievement

O padrão utilizado para este mapeamento é o Padrão Dependent Mapping porque a entidade Achievement depende semanticamente de das entidades Tournament e Club, ou seja, não pode existir como *standalone*.

```
@Entity
@Table(name = "achievements")
public class Achievement {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long achievementId;
```

Define a *primary key* da entidade Achievement. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @Column(name = "placement")
    private int placement;
```

O atributo Placement é mapeado com a anotação @Column e é do tipo primitivo int porque só guarda a posicao de uma equipa num torneio.

```
    @ManyToOne
    @JoinColumn(name = "tournament_id", nullable = false)
    private Tournament tournament;
```

O atributo Tournament foi mapeado com recurso à anotação @ManytoOne uma vez que um Achivement só pode ocorrer num e num só torneio mas um torneio pode (e tem) vários Achivements.
A anotação @JoinColumn define a coluna que é usada para unir as duas entidades. Anotamos também a propriedade nullable = false porque um Achivement nunca pode existir sem ter um torneio associado.

```
    @ManyToOne
    @JoinColumn(name = "club_id")
    private Club club;
```

A Anotação @ManyToOne é na nossa opinião a única que faz sentido uma vez que um clube pode ter vários Achivements, mas como mencionado acima, um Achivement só pode estar vinculado a um clube.

Mapeamento da entidade Card

O padrão utilizado para este mapeamento é o Padrão Dependent Mapping porque a entidade Card depende semanticamente de das entidades CardType e Player, ou seja, não pode existir como *standalone*.

```
@Entity
@Table(name = "card")
public class Card {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long cardId;
```

Define a *primary key* da entidade Card. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @ManyToOne
    @JoinColumn(name = "player_id", nullable = false)
    private Player player;
```

Usamos a anotação @ManyToOne pois neste caso a um jogador podem ser assinalados vários cartões mas esse cartão só pode ser associado a ele. A anotação @JoinColumn especifica o nome da coluna na base de dados que será usada como *foreign key* para ligar esta entidade à tabela Player.

```
    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private CardType cardType;
```

Este atributo foi mapeado com a anotação @Column e contém somente um enum com o tipo de cartão respetivo. O nullable = false foi utilizado pois obriga a que o tipo de cartão seja sempre especificado.

```
    @ManyToOne
    @JoinColumn(name = "match_statistics_id")
    private MatchStatistics matchStatistics;
```

Match Statistics foi mapeado com uma anotação @ManyToOne e é uma *foreign key* que indica que cada instância da entidade Card está associada a uma única instância de MatchStatistics. Isto é, existe a possibilidade de vários cartões serem associados a um MatchStatistics mas o contrário não se verifica(um cartão só pode estar associado a um MatchStatistics)

Mapeamento da entidade Club

```
@Entity
@Table(name = "clubs")
public class Club {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long clubId;
```

Define a *primary key* da entidade Goal. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @Column(name = "name")
    private String name;
```

```
    @OneToMany(mappedBy = "club")
    private List<Team> teams;
```

Esta associação expressa que um Club pode ter várias Teams associadas. O mappedBy = "club" indica que o lado dono da relação é a entidade Team e por isso a *foreign key* está Na Team no atributo club.

```
    @Transient
    private List<Match> matchHistory;
```

A anotação @Transient impede que a MatchHistory seja mapeada para a base de dados. Sentimos que não havia necessidade de mapear esat informação permanentemente na base de dados e por isso mapeamos assim.

```
    @OneToMany(mappedBy = "club")
    private List<Achievement> achievements;
```

A anotação @OneToMany indica que um Club pode ter vários Achievements. O mappedBy = "club" faz com que a entidade Achievement contém a chave estrangeira club_id, sendo o lado dono da relação.

```
    @ManyToMany(mappedBy = "clubs")
    private List<Tournament> tournaments;
```

O padrão utilizado para este mapeamento é o Association Table Mapping porque é o único, na nossa opinião, que permite mapear com clareza as entidades clubs e tournaments, já que um clube pode participar em vários torneios mas um torneio também tem vários clubes.

Mapeamento da entidade Goal

O padrão utilizado para este mapeamento é o Padrão Dependent Mapping porque a entidade Goal depende semanticamente de das entidades Match e Player, ou seja, não pode existir como *standalone*.

```
@Entity
@Table(name = "goals")
public class Goal {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long goalId;
```

Define a *primary key* da entidade Goal. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @ManyToOne
    @JoinColumn(name = "player_id", nullable = false)
    private Player player;
```

```
    @ManyToOne
    @JoinColumn(name = "match_id", nullable = false)
    private Match match;
```

```
    @ManyToOne
    @JoinColumn(name = "match_statistics_id")
    private MatchStatistics matchStatistics;
```

Mapeamento da entidade Match

```
@Entity
@Table(name = "matches")
public class Match {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long matchId;
```

Define a *primary key* da entidade Match. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @ManyToOne
    @JoinColumn(name = "principal_referee_id", nullable = false)
    private Referee principalReferee;
```

Usamos a anotação @ManyToOne porque é esperado que um árbitro possa ser o principal em vários jogos, mas cada jogo só pode ter um árbitro principal. O nullable = false garante que toda partida tenha um árbitro principal definido, mesmo que hajasó um arbitro, esse é sempre o árbitro principal.

```
    @ManyToMany
    @JoinTable(
            name = "users_matches",
            joinColumns = @JoinColumn(name = "match_id"),
            inverseJoinColumns = @JoinColumn(name = "user_id"))
    private List<Referee> referees;
```

Como vários árbitros podem estar envolvidos numa partida e cada árbitro pode participar de várias partidas, é necessário um relacionamento @ManyToMany, com uma tabela de união (users_matches) que contem as associações por id. Através da anotação @JoinTable criamos a tabela intermédia que vai conter os ids dos Referees e dos Matches e damos-lhe o nome de user_matches.

```
    @ManyToOne
    private Team team1;

    @ManyToOne
    private Team team2;
```

Como uma equipa participa em várias partidas e cada partida envolve exatamente duas equipas, o relacionamento é mapeado com uma relação @ManyToOne que invoca a *primary key* da equipa como *foreign key* deste campo.

```
    @Column (name = "date")
    private LocalDate date;

    @Column (name = "time")
    private LocalTime time;
```

Estes dados representam o dia e a hora que o jogo ocorre. São dois objetos fornecidos por java.time, mapeados diretamente para colunas da tabela matches. A separação entre data e hora ocorrem por uma questão de facilidade de interpretação dos dados na base de dados bem como para atender eventuais necessidades futuras de separação.

```
    @ManyToOne
    @JoinColumn(name = "stadium_id")
    private Stadium stadium;
```

O estádio foi mapeado com recurso a uma relação @ManyToOne uma vez que um estádio pode receber várias partidas ao longo do tempo, mas um jogo só ocorrem em um estádio.

```
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "stats_match_statistics_id")
    private MatchStatistics stats;
```

Este mapeamento representa as estatísticas singulares de um jogo, nesse sentido a anotação usada foi @OneToOne porque existe uma relação de um para um entre a partida e suas estatísticas. O cascade = CascadeType.ALL garante que ao persistir ou apagar uma partida, as estatísticas associadas sejam também persistidas ou apagadas.

```
    @ManyToOne
    @JoinColumn(name = "tournament_id", nullable = true)
    private Tournament tournament;
```

Associa um jogo a um torneio. Como um torneio pode conter vários jogos e cada jogo pertence a no máximo um torneio, o relacionamento que consideramos adquado é @ManyToOne. Nesse sentido, nullable = true assegura que jogos fora do contexto de torneio (jogos amigáveis, por exemplo) também são válidos.

Mapeamento da entidade MatchStatistics

```
@Entity
@Table(name = "match_stats")
public class MatchStatistics {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long matchStatisticsId;
```

Define a chave primária da entidade MatchStatistics. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
        @OneToOne(mappedBy = "stats")
        private Match match;
```

```
        @Column (name = "team1_score")
        private int team1_Score;

        @Column (name = "team2_score")
        private int team2_Score;
```

Mapeamos os golos marcados pelas equipas com recurso ao tipo primitivo int guardados numa coluna uma vez que nos pareceu a forma mais simples, intuitiva, mas também eficaz de mapear este dado.

```
        @OneToMany(mappedBy = "matchStatistics", cascade = CascadeType.ALL)
        private List<Card> team1_Cards;

        @OneToMany(mappedBy = "matchStatistics", cascade = CascadeType.ALL)
        private List<Card> team2_Cards;

        @OneToMany(mappedBy = "matchStatistics", cascade = CascadeType.ALL)
        private List<Goal> goals;
```

Neste caso, a solução mais intuitiva seria mapear estes dados com recurso a um HashMap. No entanto, durante os exemplos dados em contexto de aula só vimos exemplo com List e por isso decidimos seguir esse mesmo padrão, que já conheciamos. Mapeamos então estas estes atributos da seguinte forma:

- Uso da anotação bidirecional `@OneToMany` (do outro lado temos `matchStatistics` com a anotação `@ManyToOne`).

- O uso de `mappedBy` indica que a entidade `Card` ou `Goal` é a dona da relação, permitindo que a *primary key* seja gerada nessas tabelas e usada aqui como *foreign key*.

```
        @Column(name = "winner_team", nullable = true)
        private Integer winnerTeam;

        @Column(name="is_over")
        private boolean isOver;
```

Neste caso usamos mapeamos a winner_team somente como um Integer (não como tipo primitivo int, mas como Integer) para que ele permita a existência de *null* uma vez que um jogo nem sempre tem vencedores, podem haver empates. Resolvemos não colocar winner_team como *foreign key* e mapear de forma mais elaborada com vista a reduzir a circularidade das relações. É possível aceder á winner_team através de match e esse sim contèm a *foreign key* para a equipa e por isso com lógica de negócio é possível estabelecer a relação.
A coluna isOver foi anotada com @Column e representa um tipo primitivo boolean que reflete o estado da partida.

Mapeamento da entidade User

```
@Entity
@Table(name = "users")
public abstract class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "user_id")
    private Long userId;
```

Define a *primary key* da entidade User. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @Column(name = "name")
    private String name;
```

Mapeamento simples do atributo name para uma coluna na base de dados

Mapeamento da entidade Player

A classe Player estende a classe User e por isso herda os atributos userId e name

```
@Entity
public class Player extends User {

    @ManyToMany
    private List<Team> teams;
```

Como vários jogadores podem estar envolvidos numa partida e cada jogador pode participar de várias equipas, é necessário um relacionamento @ManyToMany, com uma tabela de união (user_teams) que contem as associações por id de ambas a entidades

```
    @Enumerated(EnumType.STRING)
    @Column(nullable = true)
    private Position preferredPosition;
```

Este atributo foi mapeado com a anotação @Column e contém somente um enum com a posicao preferida do jogador. O nullable = false foi utilizado pois obriga a que a posicao favorita do jogador não possa ser nulável e por isso seja sempre especificada.

Mapeamento da entidade Referee

A classe Referee estende a classe User e por isso herda os atributos userId e name

```
@Entity
public class Referee extends User {

    @Column(name = "has_certificate")
    private boolean hasCertificate;
```

Mapeamento simples de um boolean hasCertificate para uma coluna na base de dados.

```
    @ManyToMany(mappedBy = "referees")
    private List<Match> matches;
```

Como vários árbitros podem estar envolvidos numa partida e cada árbitro pode participar de várias partidas, é necessário um relacionamento @ManyToMany, com uma tabela de união (users_matches) que contem as associações por id.

Mapeamento da entidade Stadium

```
@Entity
@Table(name = "stadium")
public class Stadium {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long stadiumId;
```

Define a *primary key* da entidade Stadium. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @Column(name = "stadium_name", nullable = false)
    private String stadiumName;
```

Este atributo foi mapeado com a anotação @Column e contém somente uma Strinh com o nome do estádio em que a partida será jogada. O nullable = false foi utilizado pois obriga a que um estádio tenha sempre um nome associado.

Mapeamento da entidade Team

```
@Entity
@Table(name = "teams")
public class Team {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long teamId;
```

Define a *primary key* da entidade Team. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @ManyToOne
    @JoinColumn(name = "club_id", nullable = false)
    private Club club;
```

O atributo Club foi mapeado com recurso à anotação @ManytoOne uma vez que uma vez que um clube pode ter vários teams mas uma team só pertence a um clube
A anotação @JoinColumn define a coluna que é usada para unir as duas entidades. Anotamos também a propriedade nullable = false porque uma Team nunca pode existir sem ter um Club associado.

```
    @ManyToMany(mappedBy = "teams")
    private List<Player> players;
```

O padrão utilizado para este mapeamento é o Association Table Mapping porque é o único, na nossa opinião, que permite mapear com clareza as entidades teams e players, já que um jogador pode ser membro de várias equipas e uma equipa tem vários jogadores

```
    @Column(name = "goalkeeper_id")
    private Long goalkeeper;
```

Mapeamento simples de Long para uma coluna na base de dados, que representa o id de um guarda-redes. Neste caso, tomamos a decisão de nao mapear de forma mais verbosa uma vez que esta informação é muito periférica e por isso, calculamos que seria consultada muito poucas vezes. Quando for necessário o seu acesso podemos fazê-lo através de lógica na camada de negócio.

```
    @OneToMany(mappedBy = "team1")
    private List<Match> matchesAsTeam1;

    @OneToMany(mappedBy = "team2")
    private List<Match> matchesAsTeam2;
```

```
    @Transient
    public List<Match> getAllMatches() {
        List<Match> all = new ArrayList<>();
        if (matchesAsTeam1 != null) all.addAll(matchesAsTeam1);
        if (matchesAsTeam2 != null) all.addAll(matchesAsTeam2);
        return all;
    }
```

Mapeamento da entidade Tournament

```
@Entity
@Table(name = "tournament")
public abstract class Tournament {

        @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long tournamentId;
```

Define a *primary key* da entidade Tournament. A anotação @GeneratedValue serve para referir essa responsabilidade para a base de dados usando a estratégia default/automática.

```
    @Column(name = "name")
    private String tournamentName;
```

Mapeamento simples de uma String com o nome do torneio para uma coluna na base de dados.

```
    @ManyToMany
    @JoinTable(
            name = "tournament_clubs",
            joinColumns = @JoinColumn(name = "tournament_id"),
            inverseJoinColumns = @JoinColumn(name = "club_id"))
    private List<Team> clubs;
```

```
    @OneToMany(mappedBy = "tournament")
    private List<Match> matches;
```

```
    @Column(name = "is_over")
        private boolean isOver;
```

Mapeamento simples de um boolean isOver para uma coluna na base de dados que determina o estado de jogo.

Mapeamento da entidade PointsTournament

```
@Entity
public class PointsTournament extends Tournament {

        @Enumerated(EnumType.STRING)
        @Column(nullable = true)
        private PointSystem pointSystem;
```