Relatório – Construção de Sistemas de Software

Grupo 5

Duarte Carvalho fc59801

Rodrigo Freitas fc59868

Tiago Lourenço fc59877

Descrição da arquitetura em camadas

A aplicação encontra-se dividida pelas camadas de apresentação, negócio e de dados.

Relativamente à camada de apresentação, a aplicação aplica o padrão MVC (Model View Controller). No caso dos controllers (anotados com @RestController), estes recebem as requisições aos endpoints e reencaminham-nas para o respetivo handler. Por exemplo, o controller do Jogador, quando recebe um pedido GET para listar os jogadores, irá passar a responsabilidade para o JogadorHandler, que fará a manipulação dos dados e retornar o resultado, que poderá ser observado, por exemplo, no swagger. A parte do Model está precisamente relacionada com os dados e as regras de negócio, que está, principalmente, representado em código através das entidades. Quanto à View, esta representa a interface com o usuário, correspondendo, neste momento, ao swagger. Para melhorar a View, será criada outra Grafic User Interface na segunda fase do projeto, usando JavaFX.

A camada principal da aplicação é a de negócio, sendo esta responsável pelos cálculos, manipulação de dados, bem como o acesso à base de dados. O principal padrão para esta camada é o Domain Model, visto que é este o responsável por organizar o código com base em conceitos do domínio, suas características e associações, possuindo as vantagens de dominar a complexidade e dividir o problema em objetos responsáveis pelo seu âmbito, contudo requer experiência em modulação e a persistência na base de dados é um desafio. Além disso, o padrão Service irá, na camada de negócio, concretizar a implementação dos enpoints

definidos pelos controllers da camada de apresentação, tal é realizado em Java através dos handlers, que possuem a anotação @Service.

Por fim, na camada de dados é aplicado o padrão Data Mapper, que separa completamente a lógica de negócio da camada de persistência de dados, feito através do ORM (Object Relational Mapper), que traduz os dados do modelo orientado a objetos para o modelo relacional. A API standard para o conceito de ORM é o JPA (Java Persistence API), que possui a sua implementação, o Hibernate. No nosso projeto, o Hibernate é usado automaticamente pelo springboot, fazendo essa conversão no "fundo dos panos", sem que nos apercebamos. Além disso, estes dados convertidos para o modelo relacional são, depois, persistidos na base de dados (em PostgreSQL), sendo essa persistência concretizada em Java através dos repositories (anotados com @Repository), por exemplo, quando se chama o método save().

Modelo de domínio

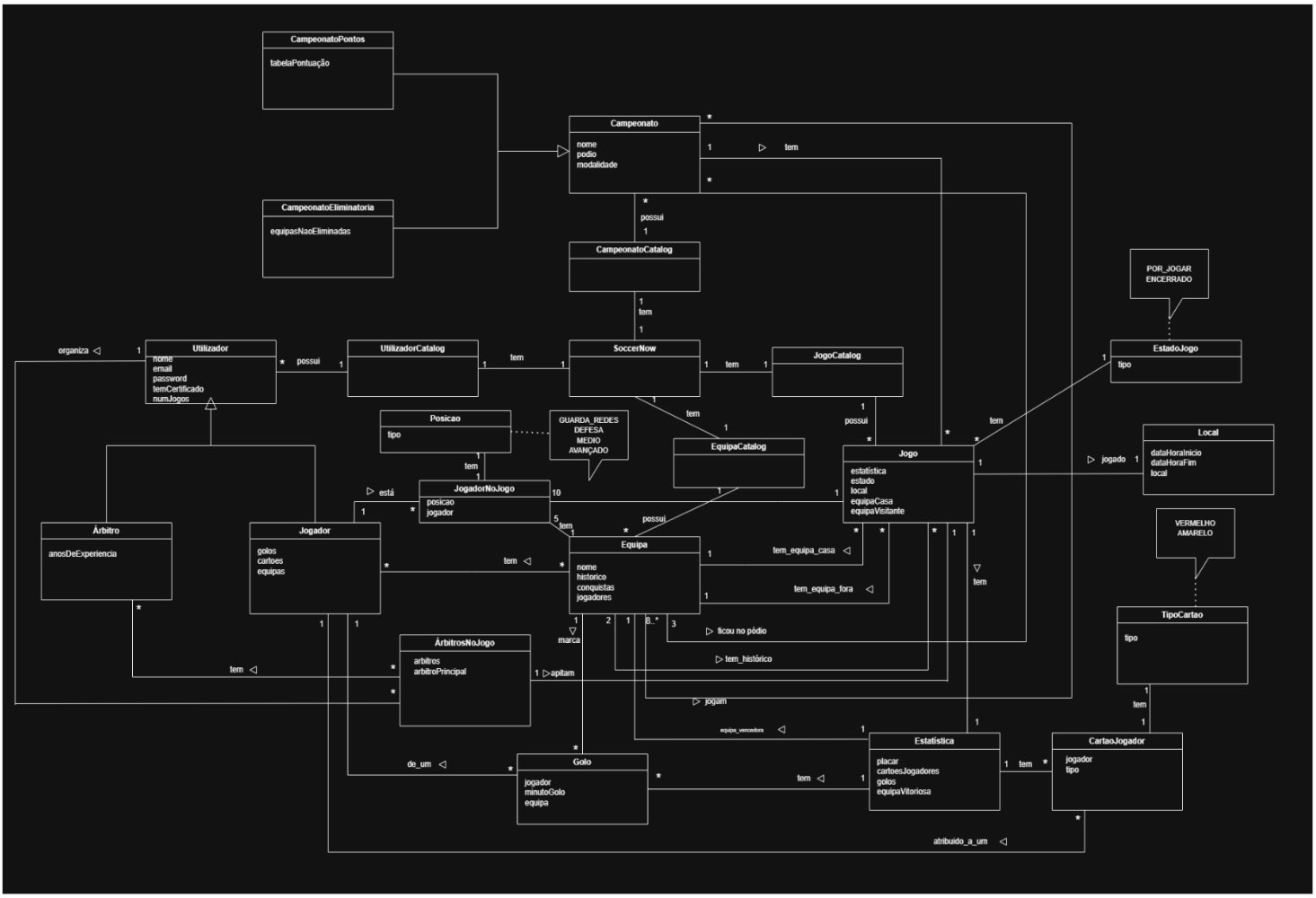
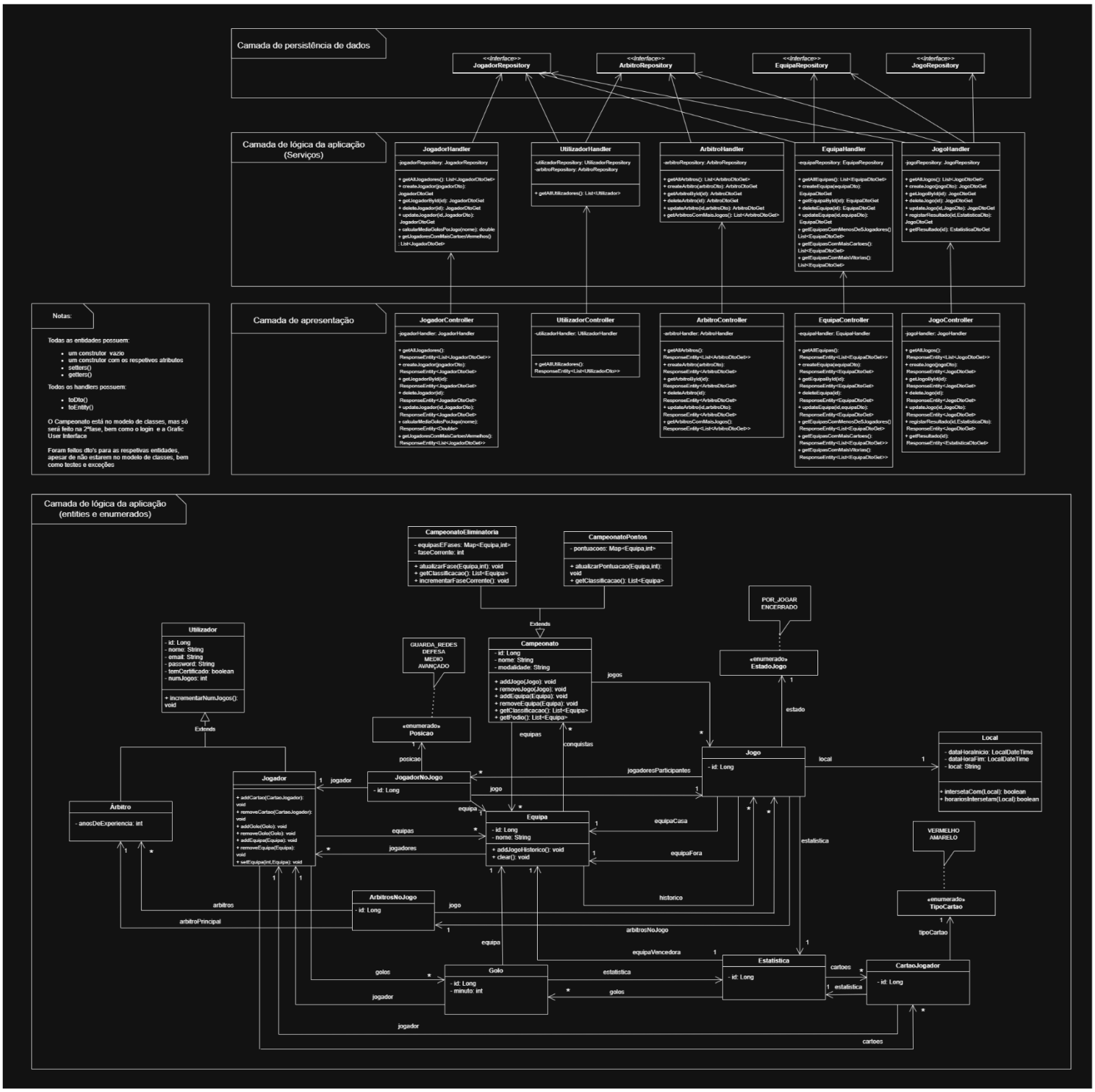
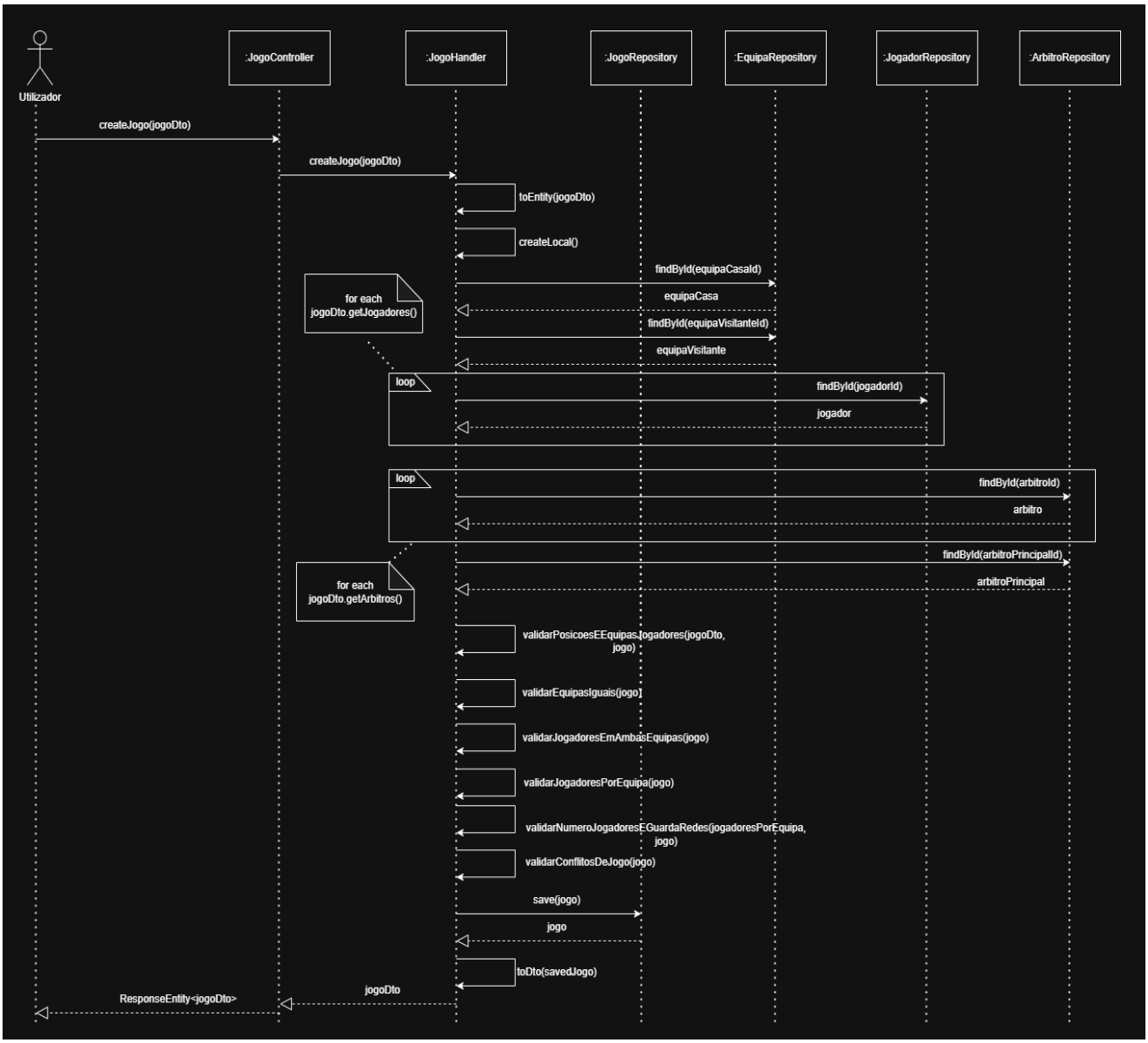


Diagrama de classes



Caso de Uso H – SSD



Mapeamento JPA

Começando com a anotação @Entity, esta é usada para indicar que uma classe é uma entidade persistente, ou seja, que será mapeada para uma tabela no banco de dados relacional. Por exemplo, se quisermos saber quais os jogadores com o nome "João", precisamos de ter a tabela Jogador na base de dados, para poder percorrer cada linha e aceder à coluna "nome", desta forma, a classe Jogador necessita de estar anotada com @Entity. O mesmo raciocínio se aplica às outras entidades que estão na pasta /entities, excepto a classe Local que é @Embeddable, mas que optámos por colocar também nessa pasta, porque será @Embedded na classe do Jogo e, consequentemente, estará presente na tabela do Jogo. O local foi feito como @Embeddable e não @Entity, pois considerámos que é uma característica inerente a um jogo e que deve aparecer diretamente na tabela do jogo de forma a simplificar a interpretação, bem como a eficiência, visto que só é preciso aceder a uma tabela, ao invés de duas. Como referido nas notas do diagrama de classes, todas as entidades têm um construtor público/protected sem argumentos para que possam funcionar corretamente.

Além disso, todas as entidades, menos as que são subclasses têm um atributo id anotado com @Id e @GeneratedValue usando a estratégia IDENTITY. Isto significa, que as entidades têm um campo imutável para a chave primária e sem significado lógico, que permite identificar univocamente cada entidade, evitando colisões de id's, visto que, é o banco de dados que controla essa geração do id automaticamente. O banco de dados usado no projeto, utiliza PostgreSQL, tendo este tipo de bancos suporte nativo para colunas IDENTITY, que é exatamente o que a estratégia IDENTITY do @GeneratedValue utiliza. Outro pormenor, é o facto de o atributo id ser do tipo Long, em vez de int, isto porque, o Long tem maior capacidade, e se o sistema crescer muito, o int pode ultrapassar o limite, além de que, muitos bancos usam tipos como BIGINT para chaves primárias por padrão, o que se alinha melhor com Long em Java.

Quando uma entidade utiliza um enumerado, este é anotado com @Enumerated (EnumType.STRING). Caso fosse utilizado EnumType.ORDINAL, os enums seriam armazenados com inteiros, e se

a ordem dos enums mudasse no código, os valores no banco passariam a apontar para o enum errado, o que não daria garantias de segurança e consistência.

Relativamente ao mapeamento de relações, para relações 1-1, foi utilizado a anotação @OneToOne, por exemplo, o jogo tem uma estatística associada, logo na classe Jogo existe o atributo estatística anotado com @OneToOne, bem como a equipaCasa e a equipaVisitante. Já para as relações 1-N, é utilizado @OneToMany e N-1 @ManyToOne. Por exemplo, um jogador pode marcar vários golos, mas um golo só é marcado por um jogador, desta forma, a classe Jogador tem o atributo golos anotado com @OneToMany e a classe Golo tem o atributo jogador anotado com @ManyToOne. Já, a relação M-N é concretizada com @ManyToMany. Veja-se o caso da equipa, que pode ter vários jogadores, bem como os jogadores, que podem estar em várias equipas. Consequentemente, a classe Equipa tem o atributo jogadores anotado com @ManyToMany, e a classe Jogador tem o atributo equipas anotado com @ManyToMany. Em qualquer um dos tipos de relações, caso a relação seja bidirecional, terá de ser usado mappedBy, de forma que não haja loops nem incosistências. Nas relações @OneToOne e @ManyToMany, ambos os lados podem ser donos da relação, podendo-se escolher um dos lados para ser o dono e o outro usa mappedBy. Contudo, nas relações @OneToMany/@ManyToOne, o lado @ManyToOne é sempre o dono da relação, portanto o mappedBy deve estar na classe @OneToMany. Por exemplo, o jogador pode ter vários cartões, mas um cartão pertence a um jogador, consequentemente, os cartões são o dono da relação, visto serem o @ManyToOne e não precisam de mappedBy, enquanto que a classe Jogador terá o atributo cartões anotado com @OneToMany (mappedBy =" jogador"). Desta forma, o cartão é o dono da relação, pois conhece o jogador, visto que possui uma coluna extra com a chave estrangeira para esse jogador. Para indicar que essa coluna contém uma foreign key, bem como o nome para essa coluna, utiliza-se a anotação @JoinColumn, que no caso do cartão, além da anotação @ManyToOne, foi também utilizado @JoinColumn (name=" jogador_id"), ou seja, a coluna extra com a chave estrangeira do jogador, irá chamar-se "jogador_id".

Para mapear herança foi utilizada a estratégia Table Per Concrete Class, em que cada classe concreta, isto é, não abstrata, tem a sua própria tabela, sendo que esta contém uma coluna por cada atributo, incluindo herdados. Foi escolhida esta estratégia, porque não tem a desvantagem do Joined Table, em que o acesso é muito lento à custa de JOINS entre tabelas, nem a do Single Table, em que há um grande desperdício de espaço, devido a existirem muitas colunas com o valor NULL, além da tabela ficar muito larga e não representar o conceito de hierarquia (está tudo numa tabela). Contudo, queries que relacionam com superclasses são custosas, porque é utilizado MERGE, ainda assim, é mais eficiente do que Joined Table (que utiliza JOIN). Em java, para aplicar a estratégia Table Per Concrete Class, basta anotar a superclasse Utilizador com @Inheritance (strategy= InheritanceType.TABLE_PER_CLASS), de forma a aplicar a estratégia Table Per Concrete Class tanto em si própria, como nas subclasses Jogador e Arbitro, que terão nas suas tabelas os atributos herdados do Utilizador, bem como os seus próprios.

Para manipular as entidades foi aplicado o padrão Repository, que fornece um objeto que é responsável por guardar e encontrar um tipo de objetos, sendo que estes se encontram na pasta /repository, por exemplo, JogadorRepository. Para pesquisas mais simples que apenas usam os atributos das classes diretamente, utiliza-se query derivation, em que não é preciso especificar com @Query, por exemplo, no jogoRepository existe o método List<Jogo> findByEstado(EstadoJogo estado), que faz a pesquisa na classe Jogo pelo atributo estado, de forma automática. Quando a busca é mais complexa, utiliza-se @Query, em que a query é feita utilizando JPQL (Java Persistence Query Language), que é uma linguagem universal que compila para vários dialectos SQL e é mais simples de que SQL, pois está mais próxima a programação orientada a objetos. Por exemplo, para responder à pergunta: em média, quantos golos os jogadores com o nome “X” marcam por jogo? foi preciso utilizar @Query no JogadorRepository, visto não ser possível fazer diretamente utilizando query derivation, ou um método default. Outro aspeto relacionado com a manipulação de entidades, é o uso de cascade. A título de exemplo, veja-se o Jogador, que utiliza cascade = CascadeType.ALL nos cartões e golos, de forma que, se o jogador for atualizado ou eliminado, os cartões e golos,

também reflitam essas mudanças. Quanto ao fetchType, foi utilizado o default, o que acaba por ser uma boa opção, visto que, é utilizado LAZY para coleções (@OneToMany e @ManyToMany), evitando carregar listas grandes automaticamente e EAGER para relações simples (@ManyTonOne, @OneToOne), presumindo que os dados são geralmente necessário juntos. Desta forma evita-se carregamentos desnecessários e menor risco de N+1 problem. Para lidar com a concorrência, foi utilizada a anotação @Version (em cima do atributo version do tipo int) que permite concorrência optimista, possuindo as vantagens de não existirem bloqueios pesados no banco de dados, evita sobrescrita de dados por transações simultâneas (se 2 pessoas alteram o mesmo registro, a versão impede que a última salve as alterações) e melhora o desempenho comparado à concorrência pessimista, uma vez que evita o uso de locks, contudo a versão não incrementa para leituras, podendo existir leituras desatualizadas.

Decisões e garantias de negócio

Além das decisões e justificações mencionadas acima, foram feitas também outras escolhas. Para começar, foram feitos dto's e dto'sGet, tal justifica-se, pelo facto de que, quando se utiliza o endpoint GET queremos obter mais informação sobre o objeto do que num POST. Por exemplo, quando se pretende criar uma equipa não faz sentido ter o atributo "histórico" disponível, mas quando se quer buscar as equipas, aí já faz sentido, para saber que jogos é que cada equipa jogou, consequentemente EquipaDtoGet tem o atributo "historico", enquanto que a EquipaDto não. A mesma ideia aplica-se aos outros dto's. Ainda relacionado com os dto's, os dto's não precisam do id, porque antes de dar POST ainda não foram criados na base de dados, mas os dto'sGet já precisam do id, de forma que se consiga identificar univocamente cada objeto. Estas características dos dto's e dto'sGet podem ser verificadas não só em código, como no swagger, uma vez que os controllers usam esses dto's. Relativamente a entidades não tão triviais de se identificar, foi criado, por exemplo, ArbitrosNoJogo, que contém os árbitros, o jogo e o árbitro principal. Esta classe serve como intermediária entre o árbitro e o jogo, sendo necessária, pois o árbitro principal varia de jogo para jogo, bem como os árbitros. Com um

raciocínio parecido, foi feita a classe JogadorNoJogo que serve de intermediária entre jogo e jogador e é necessária, pois não são todos os jogadores da equipa que jogam, além de ser uma forma de saber para o jogo X quais foram os jogadores que jogaram. A classe estatística com os golos, cartões, equipa vencedora também foi criada, de forma a que cada jogo possua uma estatística associada, e seja possível saber, por exemplo, quem ganhou, quantos golos marcou o jogador/equipa X, quando cartões levou o jogador/equipa X, etc.

Adicionalmente, foram adicionadas exceções de forma a garantir as regras de negócio e que não haja situações sem sentido. Tais exceções foram colocadas na pasta /exceptions. Foram criadas várias exceções, mas uma das mais importante, é o facto de um jogo não poder ser marcado para um horário e local, se já existir um jogo marcado para um horário que intersete esse horário e que seja no mesmo local. Para tal, foi criada a exceção JogoConflitanteException, que será verificada no jogoHandler no método de criação de jogo. Para garantir que a aplicação funciona corretamente, foram feitos testes que testam tanto o funcionamento normal, como as exceções. Estes testes encontram-se no src/test/java. Além dos testes, pode-se utilizar o swagger, para também verificar a execução da aplicação, e que está programado para, caso haja uma exceção, retorne uma mensagem personalizada, de forma que o utilizador se aperceba do erro, tal é feito em Java através do GlobalExceptionHandler que se encontra na pasta /exceptions.