Relatório – Fase 2- Construção de Sistemas de Software

Grupo 5

Duarte Carvalho fc59801

Rodrigo Freitas fc59868

Tiago Lourenço fc59877

Nota: devido ao tamanho não foi possível dar push dos vídeos, desta forma, os mesmos encontram-se disponíveis nos links abaixo:

Vídeo Thymeleaf: clique aqui para vídeo do thymleaf

Vídeo JavaFX: clique aqui para vídeo do javafx

Nota: No vídeo, ao atualizar o campeonato, apenas atualizei os jogos, mas também poderia ter feito uma atualização mais trivial como mudar o nome (como no jogador, por exemplo) ou a modalidade. Caso o nome fosse igual ao do outro campeonato, seria mostrada mensagem de erro para o utilizador.

Descrição da arquitetura do projeto

A aplicação segue uma arquitetura em três camadas: apresentação, negócio e dados, permitindo uma clara separação de responsabilidades e facilitando a manutenção e escalabilidade do sistema.

Estrutura Geral

- **Camada de Apresentação (Frontend)**
    - Esta camada é composta por duas interfaces:
        - **Interface Web** (baseada em Thymeleaf): funciona no modelo tradicional cliente/servidor, em que o cliente

envia pedidos HTTP e o servidor gera as páginas HTML com base nesses pedidos.

- **Interface JavaFX** (aplicação desktop): trata-se de uma aplicação rica (rich client), com lógica de apresentação local e comunicação com o backend via chamadas REST.

- Thymeleaf (Web App) utiliza Server-Side Rendering (SSR)
  - O HTML é gerado no servidor com base em templates (Thymeleaf) e enviado ao cliente.
  - Cada navegação (por exemplo: clicar num link) exige uma nova requisição ao servidor, que devolve uma nova página HTML inteira.
  - Vantagens:
    - URLs distintas por página (bom para SEO, acessibilidade e partilha de links).
    - Mais simples de implementar sem JavaScript complexo.
  - Desvantagens:
    - Navegação menos fluida; há recarregamento completo da página.

- JavaFX (Rich Client Application) é uma aplicação nativa cliente, fora do espectro SSR/SPA
  - JavaFX não é uma aplicação Web, mas sim uma aplicação desktop que corre no cliente.
  - Toda a interface é local, e a comunicação com o servidor (backend Spring Boot) é feita via REST APIs.
  - A lógica e o estado estão no cliente, sem recarregamentos de interface, semelhante a uma SPA.

        - Vantagens:
            - Rápida, fluida e independente do browser.
            - Total controlo da interface e interatividade.
        - Desvantagens:
            - Requer distribuição do executável.
            - Não é acessível via browser (ao contrário de web apps).
    - Ambas as interfaces partilham o mesmo backend, acedendo aos dados e lógica de negócio através de uma API RESTful comum, promovendo reutilização e coesão.

- **Camada de Negócio**
    - Esta camada contém as regras de negócio, orquestra operações complexas e expõe os serviços da aplicação através de endpoints REST.
    - É responsável por validar regras, aplicar lógica e coordenar a manipulação de dados, sendo o núcleo funcional do sistema.

- **Camada de Dados**
    - Responsável pela comunicação com a base de dados relacional (PostgreSQL), utilizando JPA/Hibernate como ORM.
    - O armazenamento, consulta e atualização de dados são feitos de forma abstrata, escondendo os detalhes de SQL da lógica de negócio.

Todas as camadas serão descritas mais detalhadamente, incluindo suporte a padrões utilizados, no próximo capítulo.

Escolha e Justificação das Decisões Técnicas na Arquitetura da Aplicação, suportadas pelos padrões desenvolvidos na aula

Observação inicial: os nomes dos padrões foram destacados a negrito, de forma a ser possível localizá-los mais facilmente.

Relativamente à camada de apresentação, a aplicação aplica o padrão **MVC** (Model View Controller) para promover a separação de responsabilidades, visto que cada componente tem uma responsabilidade distinta, promovendo escalabilidade e manutenção.

Este padrão caracterizado por:

- Modelo (Model): Representa os dados no MVC. Implementa a lógica de negócio e o processamento dos dados (garante estado e integridade), é responsável por responder a pedidos e comandos do controlador e notifica a vista sobre mudanças

- Vista (View): Apresenta os dados ao utilizador, mostra a informação com base no modelo, encaminha interações do utilizador para o controlador, estando separado da regra de negócio. Implementada em duas frentes — Web (Thymeleaf), com HTML/CSS e JavaFX (aplicação desktop), com ficheiros fxml/CSS para a vista.

- Controlador (Controller): Contém a lógica de controlo que intermedeia a comunicação entre a View e o Model, tratando eventos do utilizador e orquestrando chamadas ao modelo, atualizando-o quando necessário. Também designado por View Controller, uma vez que manipula a View e capta/ reencaminha ações do utilizador, como cliques em botões.

Para JavaFX os controllers encontram-se na pasta “controller” e as vistas na pasta “resources”, correspondendo aos ficheiros fxml. Já o model pode ser dividido em 2 partes:

- Model no frontend **(**JavaFX**)**:

    - São classes como, por exemplo, o Campeonato com propriedades Property (ex: StringProperty, IntegerProperty, etc.).
    - Servem para ligação de dados (data binding) com os elementos da interface gráfica (TableView, Label, etc.)
    - Facilitam a atualização automática da interface quando os dados mudam.
    - São independentes do backend, porque a aplicação JavaFX é uma aplicação rica (rich client) que possui lógica própria e manipula dados localmente, mesmo antes ou enquanto interage com o backend.

- Model no backend **(**Spring Boot**)**:

    - São as entidades JPA, como @Entity Campeonato
    - Representam os dados que existem na base de dados e são usados nos serviços REST (dto's, serviços, repositórios, etc)

Para a aplicação Web (Thymeleaf), o controller corresponde ao WebController da pasta "controller" e as vistas, aos ficheiros html presentes na pasta "resources/templates". Quanto ao model, neste caso:

- A aplicação Web não possui modelos próprios como o JavaFX, porque ela depende do backend para toda a lógica e dados.
- A Web App (com Thymeleaf) é baseada no paradigma cliente-servidor tradicional, em que quase toda a lógica reside no servidor.
- Assim, usa diretamente os modelos do backend (entidades, DTOs, objetos de serviço), que são passados pelo controller como atributos no Model do Spring.

Outro padrão utilizado na camada de apresentação foi o **Observer,** em que são usados eventos e listeners para comunicar ações de input e também para a comunicação entre a view e o (view)model (por exemplo, se clicar no botão atualizar equipa, é acionado um evento de clique que chama um método no controlador, o qual modifica os dados do modelo (Equipa). Como as propriedades do modelo são observáveis (ex: StringProperty nome), quaisquer alterações são automaticamente refletidas nos elementos visuais associados, como Label, TextField ou TableView.

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().

Além dos padrões mencionados acima, forma também utilizados os seguintes padrões nas camadas de negócio/dados:

- **Injeção de Dependência (Dependency Injection):** A utilização de Springboot permite a fácil gestão de componentes e promove o baixo acoplamento entre os módulos da aplicação. Este padrão é concretizado em código, por exemplo, anotando com @Autowired os repositórios injetados nos handlers.
- **Embeddable Value:** o Local que é @Embeddable e 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.
- **Dependent Mapping:** A classe dona trata do mapeamento dos seus dados/objetos que agrega. Por exemplo, um campeonato tem vários jogos (campeonato dependente dos jogos), logo irá tratar do mapeamento JPA dos jogos, utilizando @OneToMany, visto ser uma relação 1-N e cascadeType.ALL, uma vez que quando se remove um campeonato, também se quer remover os jogos associados.
- **Foreign Key Mapping:** Uma referência para outra entidade é mapeada para uma Foreign Key. Por exemplo, uma equipa possui várias conquistas (@OneToMany), logo cada conquista tem uma foreign key para a equipa, sendo essa referência localizada na coluna equipa_id da tabela das conquistas e relativa ao id da equipa.
- **Association Table Mapping:** Cria-se uma tabela para fazer mapeamento de M-N. Veja-se a equipa que possui o atributo List<Jogo> histórico, anotado com @ManyToMany. Desta forma é criada uma tabela auxiliar intermediária, com as colunas equipa_id e jogo_id, para saber que jogos cada equipa fez.
- **Repository:** Fornece um objecto que é responsável por encontrar e guardar um tipo de objectos. Toma o papel do catálogo que foi visto em DCO, mas com persistência. No projeto isso é claro, ao usar

repositórios para jogadores, árbitros, equipas, jogadores, jogos e campeonatos.

- **Unit of Work:** Mantém uma lista de objectos afectados/managed (regista novos objectos, objectos alterados ou objectos que devem ser apagados), permitindo fazer commit (e rollback) das alterações todas. Por exemplo, tudo o que for feito dentro de algo anotado com @Transactional (usado nos testes) será agrupado numa transação, ou seja, é a unidade de trabalho que pode ser commitada ou revertida (rollback) automaticamente. Desta forma, cada método de teste corre dentro de uma transação. Após a execução do método, a transação é automaticamente revertida (rollback). Isto garante que os dados inseridos ou modificados não afetam os testes seguintes, mantendo o estado da base de dados sempre limpo.
- **Identity Field:** cada objeto tem um campo que representa diretamente a sua identidade (ID), geralmente mapeado para a primary key (PK) da tabela na base de dados. Consequentemente, 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.
- **Lazy e Eager Load:** 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 (@ManyToOne, @OneToOne), presumindo que os dados são geralmente necessários juntos, evitando-se carregamentos desnecessários e menor risco do problema N+1.

Relativamente às decisões técnicas da camada de apresentação, estas serão descritas nos próximos 2 capítulos.

Quanto às decisões técnicas sobre a camada de negócio, 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 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 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. Para a fase 2, foi adicionada a lógica do campeonato à semelhança do que foi feito na fase 1, ou seja, foi criado o controller, handler, repositório, dto e entity do campeonato. Como deve ser possível cancelar um jogo de campeonato, no enumerado EstadoJogo foi adicionado o tipo CANCELADO. Consequentemente, foi adicionada a exceção de não ser possível registar o resultado de um jogo cancelado. Além disso, sem considerar as exceções triviais como a de não poderem haver campeonatos com o mesmo nome ou o campeonato deve ter pelo menos 8 equipas, foram adicionadas as exceções para verificar se os jogos de campeonatos têm todos pelo menos um árbitro certificado, além de não ser possível eliminar um campeonato em andamento. Outra exceção que adicionámos foi a de não ser possível editar um campeonato já encerrado, pois, caso contrário, caso uma equipa X tenha ganho um campeonato de futebol e fosse alterada a modalidade para voleibol, não seria consistente nem faria sentido. Ainda sobre os campeonatos, além do campeonatoDto usado para POST, foi criado campeonatoDtoGet, para, caso se queira observar os campeonatos, serem mostrados atributos “extra”, como as pontuações, número de jogos totais e número de jogos realizados.

Escolha e Justificação das Decisões Técnicas no Desenho da Interface Web

A interface Web foi desenvolvida com base no padrão Spring MVC, em conjunto com o motor de templates Thymeleaf, com foco em simplicidade, organização e responsividade. As principais decisões técnicas incluem:

- WebController: esta classe centraliza os endpoints de navegação da aplicação Web. Atua como um adaptador entre os controladores REST (como JogadorController, EquipaController, etc.) e as páginas HTML da interface, que se encontram em resources/templates, ou seja, ela serve como ponte entre os

dados do backend e a camada de apresentação HTML com Thymeleaf.

- De forma geral, cada página corresponde a um filtro. Além disso, as tags HTML têm atributos que começam com o namespace “th”, de thymeleaf. Estes serão processados no servidor e nunca chegarão a ser vistos pelo browser, nem no ver código fonte. Os mais usados foram: “th:each”, usado para percorrer uma lista, e “th:if” para blocos condicionais.

- As páginas que não correspondem a um filtro concreto são as seguintes: login, menu inicial e registar resultado.

- Botão para voltar para trás em todas as páginas menos login e menu inicial, de forma a facilitar a navegação.

- Na página de registar resultado, apenas é possível selecionar os jogos que ainda estão por jogar, e ao clicar num jogo, caso se queira adicionar um golo ou cartão, os jogadores disponíveis no dropdown são apenas os relativos a esse jogo, tal foi concretizado através de javascript.

- Quanto ao estilo, em cada página html foi aplicado css diretamente no próprio html. Em concreto foi usado: <link rel="stylesheet"href="https://cdn.jsdelivr.net/npm/@picocss/pico@1/css/pico.min.css">

- Separação clara entre lógica e apresentação: Toda a lógica da aplicação está concentrada nos controladores Spring (como WebController), enquanto os ficheiros HTML em resources/templates (com Thymeleaf) tratam apenas da

apresentação. Assim, o frontend é responsável por exibir os dados e capturar inputs, e o backend trata da sua lógica.

- Por motivos de segurança e confidencialidade as passwords não foram expostas no frontend

- Renderização do lado do servidor (SSR - Server-Side Rendering): Cada interação (como submissão de formulário ou carregamento de filtro) resulta numa nova renderização da página no servidor. Essa abordagem foi escolhida por:

    - Garantir URLs estáveis e semânticos.
    - Facilitar a partilha e indexação dos conteúdos (bom para SEO).
    - Reduzir a complexidade do frontend, sem dependência de frameworks SPA como React ou Angular.

- Uso do modelo ViewModel (Spring Model): Os controladores populam os modelos com objetos simples (DTOs como EquipaDtoGet ou JogadorDtoGet), que são automaticamente integrados nas views através de Thymeleaf (exemplo: <li th:each="e : ${equipas}" th:text="${e.nome}">).

- Comunicação RESTful (HTTP GET): Embora as chamadas à API REST backend também estejam disponíveis, grande parte da interface Web utiliza controladores diretos (Spring MVC) em vez de JavaScript/AJAX, simplificando o desenvolvimento e depuração.

Esta abordagem não corresponde a uma Single Page Application (SPA), mas segue uma filosofia semelhante ao oferecer várias páginas com

filtros e interações dinâmicas. A decisão de não utilizar uma framework SPA completa foi motivada pela necessidade de manter a complexidade reduzida, garantir compatibilidade com os padrões Spring e focar nos objetivos pedagógicos do projeto.

Escolha e Justificação das Decisões Técnicas no Desenho da Interface JavaFX

A interface JavaFX foi desenvolvida como uma aplicação rich client nativa, com foco em interatividade imediata, fluidez na experiência do utilizador e maior autonomia do lado do cliente. As principais decisões técnicas adotadas incluem:

- **Separação entre lógica e apresentação com FXML e Controladores Java**: A interface gráfica é definida em ficheiros fxml, localizados na pasta resources, permitindo separar claramente o desenho da UI da lógica de negócio. Cada vista está associada a um controlador Java na pasta controller, seguindo o padrão MVC. Isso facilita a manutenção e reutilização das interfaces, além de reduzir a complexidade do código.

- **Uso de propriedades observáveis no Model** (frontend): Ao contrário da aplicação Web, a aplicação JavaFX mantém os seus próprios modelos no frontend (model), com propriedades do tipo StringProperty, IntegerProperty, ObservableList, etc. Estes modelos permitem data binding com os elementos da interface, garantindo que alterações nos dados se reflitam automaticamente na interface (e vice-versa), sem necessidade de atualizações manuais.

- **Estilização com CSS aplicado globalmente**: Um ficheiro .css (init.css) é associado às cenas da aplicação, garantindo uma

aparência uniforme. Os estilos incluem cores, tamanhos, fontes, e foram aplicados a botões, tabelas, caixas de texto, etc., proporcionando uma interface moderna e coerente.

- **Verificações no frontend**: Verificações mais rápidas como nome vazio, foram verificadas diretamente no frontend, sendo lançada uma mensagem personalizada. Desta forma, o servidor não chega a ser chamado, evitando sobrecarga no mesmo.

- **Verificações no backend:** Exceções menos triviais foram tratadas no backend e a mensagem da exceção foi apanhada e mostrada no frontend para o utilizador, daí aparecerem exceções no terminal do vscode no vídeo de javafx. Por exemplo, exceção de já haver jogo marcado no horário e local.

- **Detalhes de um jogo:** Quando se quer ver os detalhes de um jogo, um dos atributos é o resultado. Contudo, o método getResultado(jogoId), irá lançar uma exceção caso o jogo ainda esteja por jogar (ou não existir). Desta forma, para buscar o resultado no frontend, se desse exceção é porque o jogo não tinha terminado, e desta forma a label "resultado" não irá aparecer, daí também aparecer exceção no terminal do vscode no vídeo de javafx.

- **Ícone**: Foi adicionado um ícone de uma bola de futebol, para tornar a aplicação mais apelativa e realista, visto que, atualmente, qualquer aplicação que se use tem ícone.

- **Função para mudar de cena**: cada página da aplicação, tem uma largura e altura personalizada, além de ter título da própria janela, feito através da função switchScene. Por exemplo, a página de criação de jogo tem maior altura e largura que as outras todas, devido ao número de campos ser maior.

- **Reutilização**: fxml foram aproveitados de forma a tornar a aplicação coerente, bem como o css que foi aplicado a todas as páginas.

- **Criação de campeonato**: na página de criar campeonato, só aparecem os jogos que estão por jogar, ou seja, jogos encerrados ou cancelados não aparecem (como se pode verificar no vídeo).

- **Função para mensagem personalizada**: função que mostra mensagem de pop-up personalizada pelo conteúdo da mensagem e cor (verde=sucesso, vermelho= erro de lógica (exceção) e amarelo=não tinha nada selecionado).

- **Participantes de jogos**: Para cada jogo, foram adicionados 2 botões para além de registar resultado: jogadores e árbitros. Em que pode ser observado os jogadores que participaram e o arbitro principal e assistente(s), respetivamente. Desta forma é possível de forma mais direta e fácil para o utilizador, por exemplo, ver quais foram os 5 jogadores de cada equipa que jogaram e os árbitros. Além disso, outra utilidade do botão "árbitros" pode ser para verificar se para esse jogo todos os árbitros não têm certificado, dessa forma, sabemos que não pode ser um jogo válido para campeonato.

- **Página de registar resultado**: apenas é possível registar resultado de um jogo por jogar (ou seja, os que não estão encerrados ou cancelados). Além disso, caso se queira adicionar um golo ou cartão, os jogadores disponíveis no dropdown são apenas os relativos a esse jogo.

- **Botões de navegação**: em cima é possível observar os botões "menu, árbitros, jogadores, equipas, jogos e campeonatos", de forma a, não só tornar a navegação mais rápida, como também

não obrigar o utilizador a voltar para o menu sempre que queira manipular/observar outra entidade. Além disso, existe o botão “sair” que, como o nome indica, encerra a aplicação.

- **Segurança**: Por motivos de segurança e confidencialidade as passwords não foram expostas no frontend