TurnGamesFw

Simplify Turn-Based Game Development

Autores: David Lopes
Nuno Bartolomeu

Orientadores: Pedro Félix
Filipe Freitas

Relatório do projeto realizado no âmbito de Projeto e Seminário
Licenciatura em Engenharia Informática e de Computadores

Junho de 2023

Instituto Superior de Engenharia de Lisboa

TurnGamesFw

Autores:

46100 David Monteiro Lopes
a46100@alunos.isel.pt

47233 Nuno Bartolomeu
a47233@alunos.isel.pt

Orientadores:

Pedro Félix
pedro.felix@isel.pt

Filipe Freitas
ffreitas@cc.isel.ipl.pt

Relatório do projeto realizado no âmbito de Projecto e Seminário
Licenciatura em Engenharia Informática e de Computadores

Junho de 2023

Resumo

O projeto **TurnGamesFW** consiste na criação de duas *frameworks*, cujo principal objetivo é simplificar a criação de jogos baseados em turnos. O jogo do galo, xadrez ou batalha naval, são bons exemplos deste tipo de jogos.

Ambas as *frameworks* fornecem ao programador ferramentas úteis para a criação de uma aplicação. Estas ferramentas, vão permitir-lhe saltar as partes de código que são comuns a todos os jogos de turno, podendo este dar prioridade às partes que fazem o seu jogo único.

A *framework* de *backend*, contem as funcionalidades do servidor, que são a criação e autenticação de utilizadores, a criação e controlo das partidas e os acessos à base de dados. A *framework* de *frontend*, contem as funcionalidades do cliente, que são o envio de pedidos e a atualização da interface. Existem ainda vários elementos da *ui* que podem ser usados na criação do site para o jogo, isto inclui a barra de navegação, a lista de jogos e o formulário para registo e para autenticação.

Uma aplicação criada com o uso das *frameworks*, poderá disponibilizar várias funcionalidades como a possibilidade de ter vários jogos diferentes disponíveis no servidor e no cliente, a possibilidade de ter várias partidas em simultâneo para jogos diferentes e a possibilidade do mesmo utilizador poder aceder à sua conta em dispositivos diferentes em simultâneo.

A API HTTP foi implementada usando *Spring-Boot* e *Spring MVC* e é responsável por processar os pedidos dos utilizadores, aplicar as regras de negócio e guardar as informações na base de dados. Para a base de dados foi escolhido o *PostgreSQL*, uma base de dados relacional.

A interface de utilizadores foi construida usando a biblioteca *React* e a biblioteca *Material UI* que fornece componentes *React* com estilos já predefinidos.

Palavras-chave: framework; backend; frontend; autenticação; Spring; Postgres; React; jogos; partidas; utilizadores; servidor; cliente.

Abstract

The **TurnGamesFW** project consists in the creation of two frameworks, whose main objective is to simplify the creation of turn-based games. Games like tic-tac-toe, chess, or battleship are good examples of this type of games.

Both frameworks provide useful tools to the programmer for the creation of an application. These tools will allow them to skip the parts of code that are common to all turn-based games, allowing them to prioritize the parts that make their game unique.

The backend framework contains the server functionalities, which include user creation and authentication, game creation and control, and database access.

The frontend framework contains the client functionalities, which include sending requests and updating the interface. There are also various UI elements that can be used in creating the website for the game, including the navigation bar, the list of games, and the registration and authentication forms.

An application created using these frameworks can offer various features such as the ability to have multiple different games available on the server and client, the ability to have multiple concurrent matches for different games, and the ability for the same user to access their account on different devices simultaneously.

The HTTP API was implemented using Spring-Boot and Spring MVC and is responsible for processing user requests, applying business rules, and storing information in the database. PostgreSQL, a relational database, was chosen for the database.

The user interface was built using the React library and the Material UI library, which provides pre-defined React components with styles.

Keywords: framework, backend, frontend, authentication, Spring, Postgres, React, games, matches, users, server, client.

Índice

Capítulo 1

Introdução

Este documento tem como objetivo reportar a realização do projeto TurnGamesFw desenvolvido no âmbito da cadeira de Projeto e Seminário 2022/23.

Neste projeto existem duas *framework* que fornecem ao programador ferramentas úteis para criar uma aplicação. Estas ferramentas servem para a reutilização de partes de código que são comuns a todos os jogos de turno, podendo assim dar prioridade às partes únicas do seu jogo. É possível, a um desenvolvedor, utilizar as funcionalidades disponibilizadas pelas *frameworks* em conjunto com objetos criados pelo mesmo, seguindo uma estrutura específica dada pelas interfaces, para criar uma aplicação onde o seu jogo possa ser jogado.

As *frameworks* mencionadas, sendo uma de *backend* e uma de *frontend*, já têm nelas implementadas as partes não especificas ao jogo. Na *framework* de *backend* estão realizadas as funcionalidades que permitem a resolução de pedidos e a criação de respostas *HTTP*, a gestão de utilizadores, a gestão de partidas e a inserção de informação na base de dados.

Já na *framework* de *frontend* estão presentes as operações que permitem a um utilizador aceder às funcionalidades do servidor, tanto ao nível de registo e autenticação, como a nível de interagir com o jogo criado. A *framework* vai ser responsável por enviar os pedidos ao servidor, atualizar a interface sempre que necessário e garantir que o utilizador está autenticado antes de o deixar aceder a páginas que necessitam de autenticação.

Neste capitulo será relatado o contexto e a motivação que levaram ao desenvolvimento deste projeto. Serão também apresentados os requisitos funcionais do projeto e a organização do documento.

1.1 Motivação

A criação de jogos é um processo árduo que envolve várias horas de conceptualização, programação e teste. Mas o grande problema reside na realidade de que muitas das funcionalidades que necessitam ser feitas não vêm do conteúdo do jogo em especifico, mas do fato de ser um jogo em si. Necessidades como criar utilizadores, guardar a informação toda na base de dados, criar diversos pedidos que vão para rotas diferentes, são apenas alguns exemplos

onde desenvolvedores precisam de investir vários recursos.

É também importante referir que o maior motivo para a não conclusão de projetos vem da diminuição de motivação ao longo da sua criação, especialmente se a parte do jogo em si não estiver a evoluir.

Por este motivo criamos a *TurnGamesFW*, que consiste em duas *frameworks* que servem como ponto de inicio para a criação de jogos baseados em turnos, ou seja jogos onde apenas um jogador interage com o jogo de cada vez. Exemplos de jogos de turno são o xadrez ou batalha naval, onde enquanto um jogador está a realizar o seu turno o outro tem de esperar.

Os desenvolvedores que utilizem as funcionalidades deste projeto, só terão de se preocupar em programar a informação especifica do seu jogo. Se usarmos os exemplos anteriores, seriam a forma como as peças se movem no xadrez ou o número de tiros por turno na batalha naval. Desta forma não vai existir um gasto adicional de tempo, por parte dos desenvolvedores para os desenvolver, isto garante um foco maior nas regras do jogo e na sua aparência, que são as partes mais interessantes de realizar no ponto de vista da aplicação.

Outro ponto que pode ser interessante para além da vantagem temporal é a vantagem a nível da estrutura. Os jogos criados irão implementar interfaces que dão evidencia aos componentes que necessitam ser desenvolvidos. O desenvolvedor que se foque nesses componentes vai obter um código modular e bem organizado que ajuda com a correção de erros e atualização de funcionalidades.

O projeto também tem vantagens em comparação com outros produtos parecidos, nomeadamente, as *frameworks* são grátis e não há necessidade de criar uma conta em lado nenhum para as utilizar e o controlo está todo do lado do programador, visto que, as interfaces não limitam o programador a um estilo de programação especifico.

1.2 Requisitos

Os requisitos necessários para o funcionamento do projeto foram:

- Definir o esquema da base de dados, para que seja possível utiliza-lo independentemente dos jogos implementados pelo programador.
- Criar a *framework* de *backend* para gestão de utilizadores, jogos e partidas.
- Criar a *framework* de *frontend* para ajudar o programador na criação da interface de utilizador.
- Criar um guia de utilização via documentação do projeto e exemplos.

Para além destes requisitos foi importante criar três jogos para experimentar as *frameworks*. Começamos por criar o "Jogo do Galo", uma vez que é extremamente simples, foi utilizado para testar o código das próprias *frameworks*. Depois cada um dos alunos focou-se num jogo mais complexo, sendo estes a "Batalha Naval"e a "Corrida no Castelo".

1.3 Organização do documento

Este documento está organizado em cinco capítulos.

- No primeiro capítulo é apresentado a introdução e as motivações por de trás da *Turn-Games Framework*. São também definidos os requisitos e a estrutura deste relatório.
- No segundo capítulo é abordada a arquitetura do sistema nomeadamente a separação da *framework* de *backend* e da de *frontend*, em conjunto com as interfaces HTTP.
- No terceiro capítulo é descrita a *framework* de *backend*, referindo o desenho da API assim como detalhes relevantes da sua implementação. Também é descrito o esquema da base de dados relacional.
- No quarto capítulo é descrita a implementação da *framework* de *frontend*. São mencionados os vários componentes criados, a forma como faz pedidos à API e como guarda a sessão do utilizador nos *cookies*.
- No quinto capítulo é dedicado à conclusão onde é dada uma visão global da realização deste projeto e dos desafios encontrados, não esquecendo as considerações finais.

Capítulo 2

Arquitetura

A arquitetura do sistema é composta por dois grandes setores, o servidor e o cliente, no lado do servidor contém a *framework backend* e a lógica de jogos, e o lado do cliente contém a *framework frontend* e os componentes dos jogos. É possível verificar esses dois setores na figura 2.1 que segue abaixo.

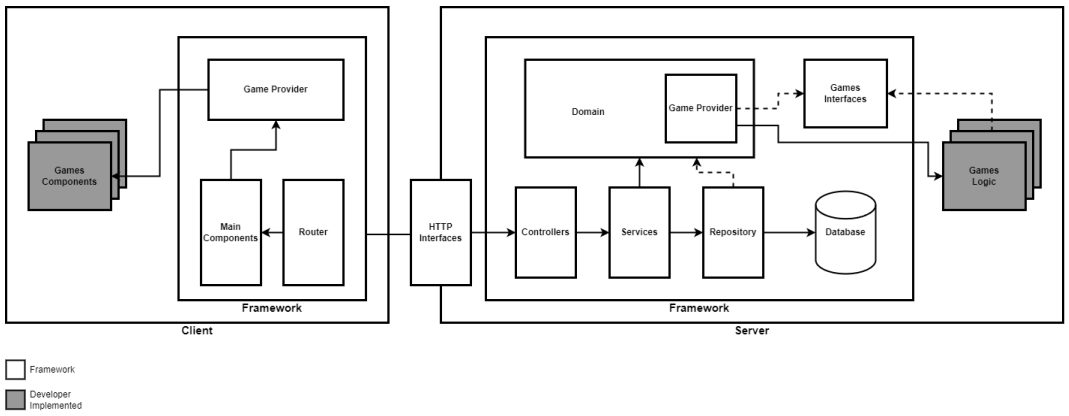

Figura 2.1: Arquitetura do Sistema

2.1 Servidor

O servidor, ou *backend*, tem como objetivo expor através dos controladores, todas as funcionalidades do sistema, executá-las, através dos serviços, e guardar todas as alterações necessárias na base de dados.

O servidor está separado em duas partes distintas, a parte já implementada que é toda a *framework*, e as lógicas de jogos que será implementado por quem usar a *framework*. Dentro da *framework* são recebido os pedidos pelos *Controllers* que fazem a delegação da execução para os *Services*, estes fazem toda a execução lógica da *framework* passando a lógica do jogo para o seu componente respetivo, sendo isso feito a partir do uso do *Game Provider*. Ainda na execução dos pedidos é usada a *Database* que serve para guardar todas as alterações feitas aos jogos e utilizadores, essa utilização é feita a partir do *Repository* que serve de meio de

comunicação entre os *Services* e a *Database*. É importante ressaltar que as lógicas de jogos são implementadas pelos desenvolvedores que usam as interfaces de jogo disponibilizadas pela *framework*. Estas mesmas lógicas são importadas/inseridas na *framework* de forma a posteriormente serem usadas pelo *Game Provider* para gestão e utilização durante a execução do servidor.

Para o servidor foi usado como linguagem de programação o Kotlin[1], fazendo também uso da *framework* Spring[2] para criação da API, ainda no servidor foi escolhido usar uma base de dados PostgreSQL[3] e usamos a biblioteca JDBI[4] para gestão da mesma. A escolha destas tecnologias foi baseada na familiarização das mesmas durante a fase académica, e a sua popularidade no mercado de trabalho.

2.2 Cliente

O cliente, ou *frontend*, tem como objetivo servir de ponto de comunicação entre o utilizador e os serviços do *backend* através da interface de utilizador.

O cliente, mais uma vez, é separado em 2 partes, a *framework*, que já está implementada com toda a interface gráfica e gestão de pedidos, e os componentes de jogos que necessitam ser implementados pelos desenvolvedores, estes representam elementos da interface gráfica e fazem a gestão das partes especificas de cada jogo.

Para gestão dos pedidos existe um *Router* que apresenta o componente correto dependendo da rota, esses componentes já estão implementados pela *framework*. Para além disso existe os componentes de jogo que necessitam implementação e são fornecidos à *framework* para assim serem usados pelo *Game Provider* para gestão e utilização dos mesmos, fazendo assim aparecer a interface adequada a cada jogo e também serem executados os pedidos para a API necessários.

Para o cliente foi usado como linguagem de programação Typescript[5], fazendo uso da biblioteca React[6] para toda a representação da interface. Usamos também ainda, para aspetos de interface, a biblioteca Material-UI[7] que contém vários componentes a serem usados para a interface de utilizador. Mais uma vez a escolha das tecnologias usadas foi a familiarização das mesmas durante a fase académica, com exceção da biblioteca Material-UI que foi aconselhada visto esta ser usada no mercado de trabalho para facilitar e agilizar a criação de interfaces de utilizador.

2.3 Interfaces HTTP

As interfaces HTTP, são todas as funcionalidades que o servidor disponibiliza assim como o modelo de dados a enviar nesses mesmos pedidos. Essas funcionalidades contam com pedidos de gestão e utilização de utilizadores, e também a pesquisa e jogabilidade dos jogos fornecidos. Para a execução correta dos pedidos existe todo um modelo de dados a ter em

conta, sendo nele passado as informações necessárias para a execução. (ver rotas e modelo em capítulo A.1).

Para o uso das rotas, em algumas é necessário o envio de um *token* que é enviado como *cookie* e representa a sessão de um utilizador, esse *cookie* é enviado usando *HTTP only* sendo ele com o nome "TGFW". Escolhemos usar *HTTP only* pela segurança visto este não ser possível de ser acedido pelo lado do cliente e ser de total controlo do servidor.

Também ainda referentes às rotas, as rotas referentes à jogabilidade do jogo não têm um *payload* definido pela *framework* e são definidas por quem implementa as lógicas de jogo, sendo assim, é preciso saber qual a definição escolhida para o envio de dados pelo lado do cliente.

Capítulo 3

Framework Backend

A *Framework Backend* visa simplificar o desenvolvimento de jogos baseados em turnos. Ela já possui toda a infraestrutura necessária para lidar com os pedidos, assim como toda a estrutura sobre os utilizadores e gestão da mesma, além disso ainda possui verificações feitas sobre dados gerais do jogo, deixando assim que os desenvolvedores se concentrem exclusivamente na criação da lógica do jogo.

O uso da *framework* consiste no fornecimento de lógicas de jogo criadas pelos desenvolvedores, fazendo uso das interfaces de jogo. Assim a *framework* cuida do gerenciamento dessas mesmas lógicas de jogo para serem usadas durante a execução dos pedidos de forma correta, sendo transmitido os dados recebidos para a lógica de jogo e posteriormente enviado como resposta ao pedido. Esses dados são específicos de cada jogo e possíveis de ser qualquer estrutura de dados pretendida.

3.1 Domínio

No domínio a estrutura de dados existente é toda a estrutura que engloba os utilizadores e jogos, da parte dos utilizadores está toda implementada pelo lado da *framework* visto não ser preciso nada adicional, essa estrutura contém os **Users** que representam toda a informação referente aos utilizadores e os **Tokens** que representam as fichas de validação de cada conexão feita por utilizador. Além de toda a parte dos utilizadores existem também na estrutura do domínio o **Game Provider** que é uma classe com objetivo de fazer toda a gestão das lógicas de jogo, nessa estrutura existe classes de domínio que representam toda a informação sobre os jogos (**Games**) e partidas (**Matches**) que não sejam específicos de um jogo (ver mais em capítulo 3.5).

Dentro da classe **Game Provider** existe uma única estrutura de dados que é um mapa de chave *String* que representa o nome do jogo, e de valor **Game Logic** que é a lógica de jogo. Existe depois um conjunto de funcionalidade que servem para fazer uso desse mesmo mapa:

3.2 Base de dados

Na *framework* está presente também uma base de dados, esta trata-se de uma base de dados relacional, SQL, têm como objetivo guardar toda a informação dos utilizadores, jogos e também partidas para assim persistir ao longo da utilização da aplicação. Nela existem 4 entidades e 4 relações, sendo essas possíveis de ver na figura 3.1 que mostra o modelo EA (entidade-associação) utilizado.

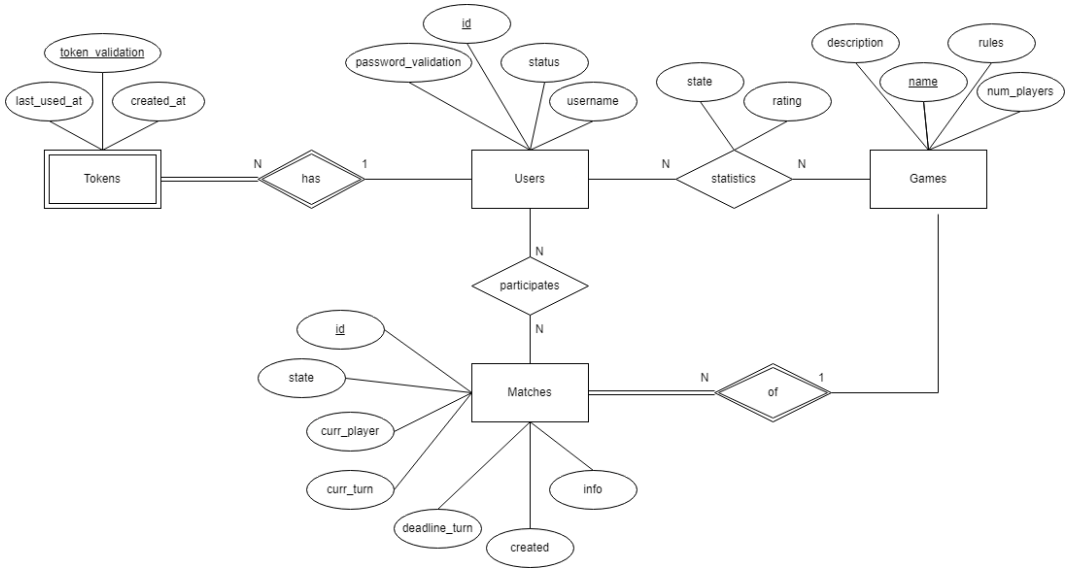

Figura 3.1: Diagrama EA.

- A entidade **Users** representa os utilizadores que irão participar nos jogos.
- A entidade **Tokens** é entidade fraca de **Users** e representa as sessões do utilizador.
- A entidade **Games** serve para representar as informações de cada jogo.
- A entidade **Matches** serve para representar as partidas realizadas. Esta entidade tem o atributo "info"que vai representar toda a informação que é especifica ao jogo.

Estas entidades foram normalizadas para 6 tabelas, com as suas respetivas colunas e relações. Na figura 3.2 está representado o esquema onde as conexões entre tabelas são feitas com o uso de chaves estrangeiras para relações 1:N e criando uma nova tabela para relações N:N.

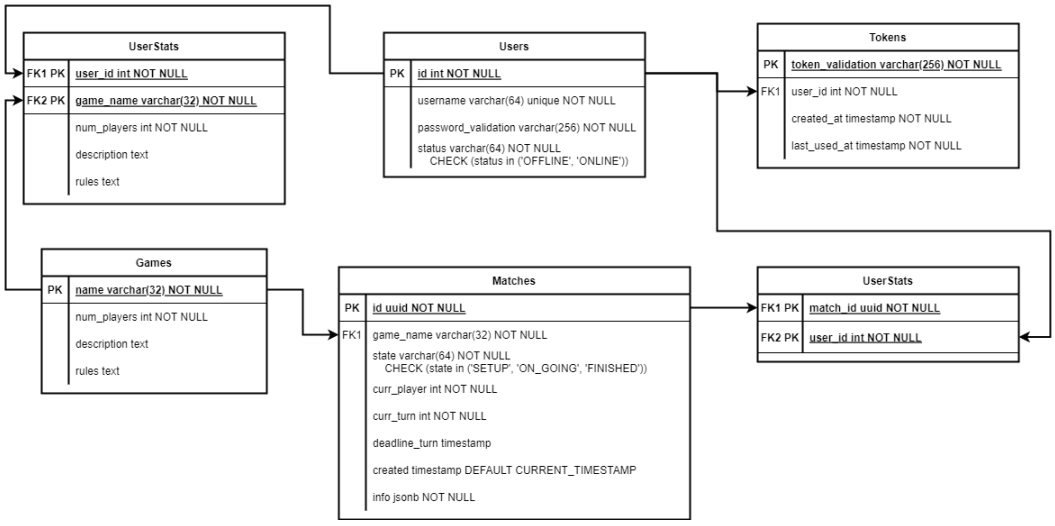

Figura 3.2: Esquema de tabelas.

As novas tabelas adicionadas a partir de relações são:

- A tabela **UserStats** que representa a informação relativa de cada jogador para um determinado jogo.
- A tabela **UserMatches** que representa os jogadores presentes em uma partida.

Para a interação da *framework* com a base de dados e todas as suas tabelas foi criado interfaces de repositório para definir todas as funcionalidades necessárias no correto uso da base de dados, isso serve para não depender de uma única escolha de implementação, que no nosso caso usamos a biblioteca JDBI[4].

3.2.1 User Repository

O **User Repository** é a interface que define todas as funcionalidades a serem feitas perante os utilizadores, isso inclui operações com a tabela **Users**, **Tokens** e **UserStats**. Nessas operações estão presentes operações de inserção de novos utilizadores, pesquisa de utilizadores por identificador e nome, existe também criação de novos tokens e pesquisa de utilizadores com base nos mesmos tokens.

3.2.2 Game Repository

O **Game Repository** é a interface que define todas as funcionalidades que são feitas perante os jogos, isso inclui operações com a tabela **Game**, **Match** e **UserMatch**. Algumas dessas operações são inserção de jogos e partidas, atualizações de partidas e obtenção de utilizadores em pesquisa de jogo.

3.3 Serviços

Os serviços são responsáveis por fazer todas as operações pretendidas, interagindo com a base de dados e fazendo uso do **Game Provider** para utilizar a lógica de jogo correta. Para uma melhor compreensão do código dividimos os serviços em dois distintos, o **User Service** que é responsáveis pelas operações sobre os utilizadores, e o **Game Service** que é responsável pelas operações sobre os jogos.

3.3.1 User Service

O **User Service**, como dito anteriormente, é responsável pelas operações sobre os utilizadores, para isso faz uso de um *Transaction Manager* para garantir a integridade no acesso à base de dados, além disso faz uso de um *Password Encoder* e um *Token Enconder* que são responsáveis por fazer a codificação da palavra passe e tokens quando necessário.

Este abrange operações de criação de utilizadores usando o *Password Enconder* para codificar a palavra passe do utilizador e inserindo na base de dados, obtenção de utilizador pelo nome e identificador fazendo uso das funcionalidades do repositório, também existem várias funcionalidades de atualização de dados de utilizador, além disso sobre os tokens é possível fazer a criação de novos tokens e é usado o *Token Enconder* para criar informações de validação e fazer a obtenção de utilizadores a partir do mesmo.

3.3.2 Game Service

O **Game Service** é responsável pelas operações sobre os jogos e suas partidas, para isso faz uso também de um *Transaction Manager* para garantir a integridade nos acessos à base de dados, e ainda faz uso do *Game Provider* para acesso às lógicas de jogo e delegação correta das funções.

As funcionalidades existentes são a criação de jogos e partidas e obtenção das mesmas, procura de jogos por utilizadores e emparelhamento de partidas, existe também as funcionalidades de jogabilidade que incluem a configuração e jogadas da partida, para essas é usado o *Game Provider* para obtenção da lógica de jogo e fazer a delegação para a função correta de jogabilidade com os dados específicos do jogo, por último existe a validação de turno do utilizador que informa se é a vez dele jogar.

3.4 Controladores

Os pedidos HTTP da *framework* estão divididos em dois controladores, um para Utilizadores (*User Controller*) e outro para Jogos (*Game Controller*). As respostas dos pedidos em caso de sucesso são enviadas em formato Siren[8] e em caso de erro enviadas em formato Problem+Json[9].

Nos pedidos existe uma estrutura de dados para o envio que consiste, nos pedidos de utilizador, no envio do nome de utilizador e palavra passe tanto para o registo como para o inicio de sessão e para os restantes pedidos de utilizador consiste no envio do token de sessão como cookie para o encerramento de sessão como para a obtenção das informações do utilizador. Agora nos pedidos de jogos existe uma estrutura e dados mas esta não é previamente definida, pelo que os dados específicos do jogo podem variar de implementação para implementação de lógica de jogo, pelo que o único valor que é estritamente necessário ser enviado nos pedidos de jogo é o cookie de sessão para garantir a sessão do utilizador e saber qual o utilizador que está fazer o pedido.

Nos pedidos que é necessário autenticação é necessário um token de sessão como referido anteriormente, esse mesmo que por sua vez passa por um intercetor do spring que é um *AuthorizationHeaderProcessor*, nele no *preHandle*, ou seja antes de ser enviado para o controller, é verificado se o cookie é enviado da forma correta com o nome "TGFW"e em seguida é feita a pesquisa de utilizador com esse mesmo token, retornando assim o *User* para o controller com o utilizador que está a executar o pedido.

3.4.1 Estados de jogo

Nos pedidos referentes aos jogos existe toda uma ordem pela qual eles devem ser executados, essa ordem descreve desde o começo de um jogo, ou seja a sua pesquisa, até ser terminado o jogo, passando por todas as ações que são possíveis de ser executadas, é possível ver a máquina de estado que representa os estados que o jogo pode tomar figura 3.3.

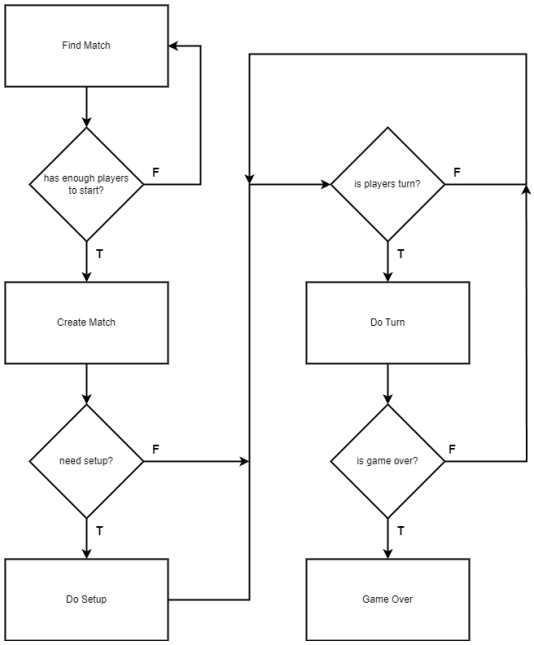

Figura 3.3: Máquina Estados Jogo.

O primeiro estado em que o jogo se pode encontrar é o "*FindMatch*", nesse estado é necessário que o jogadores façam a procura de uma partida, quando um jogador executa essa função é verificado se já existe jogadores suficientes para ser criada uma partida, caso contrário eles encontram-se em lista de espera sendo necessário que outro jogador procure uma partida, para ambos prosseguirem para a próxima fase. No próximo estado a partida é criada e com isso retornada para os jogadores que estavam à procura de jogo, em seguida é verificado se o jogo têm fase de "*Setup*"e caso haja essa fase é necessário que os jogadores executem a função de configuração, sendo esta assíncrona, ou seja, é possível ser executada em simultâneo. No fim da fase de "*Setup*"quando todos os jogadores tiverem executado ou em caso desta não existir, o jogo encontra-se em fase de andamento, nesta fase é verificado se é a sua jogada ou a vez do oponente, em caso de ser a sua jogada é executada a função de "*DoTurn*"que executa a jogada do utilizador, por último no fim de cada jogada é verificado se o jogo se encontra terminado, caso se encontre ambos os jogadores recebem o jogo como terminado não sendo possível ser executado mais nenhum função sobre o mesmo.

3.5 Interface de jogos

Sendo necessário a implementação externa por desenvolvedores para as lógicas de jogo existe toda uma interface de jogos para a criação das mesmas, para isso como estrutura de dados foram definidas as classes **Game** e **Match**, uma para representar os jogos e outra para representar as partidas. Estas contêm campos de informação genérica para todos os jogos e um campo de informação específica do jogo com o objetivo de guardar qualquer tipo de estrutura de dados pretendida. Para além disso, existe ainda, a interface que representa uma lógica de jogo, que consta com as funcionalidades necessárias a ser implementadas e ainda estruturas de dados de receção e envio de dados para a *framework*, essa interface chama-se **Game Logic**.

3.5.1 JsonNode

Como referido anteriormente existe dados específicos de cada jogo que necessitam ser guardados, tanto para a *framework* como para a base de dados. Sendo esta uma tarefa difícil de conseguir guardar os dados de forma que possam ser estruturas grandes e muito específicas, decidimos guardar esses mesmo dados em um campo do tipo **JsonNode** que representa um objeto no formato Json[10], desta forma conseguimos guardar facilmente, enviar e receber esses mesmos objetos sem ter conhecimento da estrutura que neles é enviado. Visto unicamente ser preciso do lado da *framework* guardar o objeto não é necessário ter o conhecimento da estrutura, e desta forma conseguimos também guardar em base de dados o objeto em formato Json. Do lado da lógica de jogo que têm conhecimento do tipo de estrutura guardada e recebida, é facilmente feita a conversão do tipo **JsonNode** para o tipo pretendido pelo

desenvolvedor.

Para o correto uso do tipo *JsonNode* na implementação de lógicas de jogo, é necessário fazer a conversão de *JsonNode* para objeto do tipo pretendido, quando recebido os dados para sua utilização, e fazer a conversão do objeto para o tipo *JsonNode* para o envio de dados nos retornos das funções para a *framework*. Para fazer essa conversão, nós usamos, nos nossos jogos, a biblioteca jackson[11], fazendo assim uso do *ObjectMapper* para converter de Json para objeto e vice versa.

3.5.2 Game Logic

O **Game Logic** é a interface de lógicas de jogos e é muito importante para a *framework*, pois é o que faz o entendimento entre a *framework* e as implementações de jogos. A interface contém um número de funcionalidades, estas que representam as funções de jogo, para além de funções de retorno de dados necessários a ser enviados aos jogadores. Essas funções constam com a criação de jogos, configuração inicial e jogadas de turno, para além disso existe o retorno de dados da vista para cada utilizador, assim como a informação geral sobre o jogo.

Para a correta implementação de uma lógica de jogo é necessário criar todas as funcionalidades presentes na interface, sendo possível não implementar a funcionalidade de configuração em caso do jogo não necessitar dessa fase. Explicando as funcionalidades e o seu uso:

- *getGameInfo* - deve retornar as informações gerais do jogo, sendo este importante para apresentação de dados de informação assim como o número de jogadores que é usado para delimitar os jogadores possíveis a entrar em uma partida.
- *matchPlayerView* - é usada para retornar a vista de cada jogador, sendo isso utíl para os casos de jogos em que não é permitida a vista completa sobre toda a informação da partida.
- *create* - é usada na criação de uma nova partida fazendo inicialização de toda a estrutura de uma partida incluindo as estruturas específicas do jogo.
- *setup* - tem como objetivo executar a configuração inicial com os dados recebidos do jogador, estes que estão na estrutura escolhida pelo desenvolvedor.
- *doTurn* - tem como objetivo efetuar a jogada de um utilizador sobre a partida, esta também recebe os dados sobre a informação da jogada na estrutura definida pelo desenvolvedor.

Sobre as funcionalidades "*setup*"e "*doTurn*", ambas fazem retorno de um tipo de dados específico que é "*UpdateInfo*", este tipo contém a informação em caso de erro, a mensagem a ser enviada ao utilizador e ainda a partida atualizada para a *framework* em caso de sucesso.

Capítulo 4

Framework Frontend

A *framework* de *frontend* foi desenhada para auxiliar os desenvolvedores a criar uma aplicação web, que permita aos utilizadores interagir com o sistema. A aplicação será executada no browser do utilizador e vai comunicar com a aplicação servidor através do protocolo HTTP, usando pedidos e respostas.

A *framework* segue a ideologia de uma *Single Page Application*, **SPA**[12]. Este método passa o processo de renderizar a página do servidor, para o próprio cliente. O cliente vai receber toda a informação necessária no primeiro pedido que realizar ao servidor. Depois desta primeira interação, cabe ao cliente apresentar, na interface, qualquer informação que o utilizador esteja a tentar acessar, sem que seja necessário interagir novamente com o servidor. Isto remove o atraso entre enviar um pedido e receber um resposta, algo que é muito importante para garantir que as atualizações da interface são discretas e realizadas em tempo real.

4.1 Rotas

As rotas representam os endereços presentes na barra de endereços do próprio browser. Elas são usadas para carregar páginas especificas que os utilizadores querem acessar.

Para fazer o controlo de rotas foi criado um objeto **RouterProvider**, disponibilizado pela biblioteca *React*[6]. Este componente mantém um *router* interno que contém todas as rotas que foram definidas na *framework*. O *RouterProvider* vai renderizar uma página, sempre que esta seja requisitada na barra de endereços, ou quando é utilizada a função *navigate("url")*. Esta função é disponibilizada pelo *React* para poder alterar a página atual por outra presente no *RouterProvider*, apenas alterando os componentes que são apresentados. Isto evita que o site seja recarregado, o que levaria a um novo pedido ao servidor.

Outro aspeto importante das rotas é a possibilidade de criar rotas variáveis. Estas rotas podem ser distinguidas quando nelas está presente o identificador **:id**. Estas rotas representam um recurso especifico que é identificado pelo *id*, por exemplo, na rota **"/user/:userId"**, dependendo do *userId*, a informação presente na página vai ser relativa ao **User** especificado.

Neste tipo de rotas foi importante adicionar uma função *loader*. Esta função vai realizar um pedido a API para obter a informação especifica do recurso, que vai ser posteriormente apresentado dentro do componente respetivo.

Na figura 4.1 está presente o diagrama com as páginas presentes na *framework* e as páginas que podem ser acedidas, a partir das mesmas.

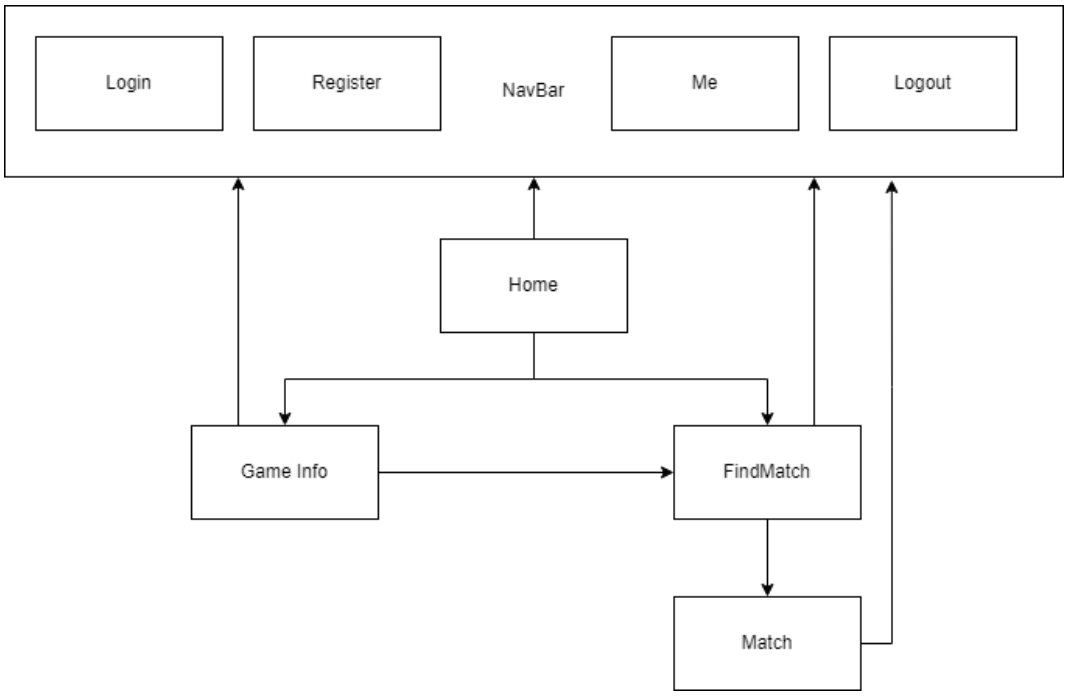

Figura 4.1: Navegação Entre Páginas.

4.2 Pedidos

Em relação aos pedidos, foi implementada na *framework* a função **fetchAPI()** que faz pedidos a API disponibilizada pelo servidor e em seguida cria as respostas. O envio do pedido é feito com uso da função *fetch()* disponibilizada pela *Fetch API*[13] para realizar os pedidos e, com um *switch case*, são avaliados os códigos de estado da resposta. Para utilizar a função *fetchAPI()* é necessário passar o *url* que está a tentar ser acedido e o método do pedido.

Adicionalmente pode ser passado o corpo do pedido, esta funcionalidade é necessária para o caso de um pedido do tipo *POST* ou *PUT* onde o servidor vai atualizar a informação registada. Também foi adicionado um booleano chamado *getBody*, que se for posto a falso o corpo da resposta é ignorado.

```
const resp = await fetchAPI(
    "/api/game/TicTacToe/turn",
    "POST",
    {
        playerId: 1
        position: [0, 2]
    },
    false
)
```

Listing 4.1: Exemplo de um Pedido

Neste exemplo é possível ver um pedido para o *url*: "/api/game/TicTacToe/turn", o pedido é do tipo *POST*, o *body* contem o id do utilizador e a posição onde está a tentar jogar, por último deixa o booleano *getBody* a falso uma vez que não necessita do corpo da resposta. Outro detalhe que pode aqui ser constatado é o uso da palavra **await**. Como o *fetchAPI* retorna um objeto do tipo *Promisse*[14] é necessário esperar que a "promessa"seja cumprida, para ter a certeza, de que o servidor já recebeu o pedido e já o resolveu.

Ainda é importante referir nesta secção a forma como é atualizada a partida. Na *framework* usamos o conceito de *Client-side Pooling*, realizando um pedido por segundo para obter a representação mais atual da partida. A esta representação é feita uma verificação do estado da mesma e do *currPlayer*, para identificar se o jogo ainda está a correr e se o turno é do jogador ou do oponente.

4.3 Componentes

Cada página, disponibilizada pela *framework*, tem um componente base que será renderizado. Nestes componentes estão presentes várias funcionalidades do *React*, nomeadamente, *states*, *hooks* e *contexts*.

- Os *states* são uma forma que o *React* tem de guardar dados internos do componente. Estes dados são mutáveis, e quando sofrem uma alteração, é possível desencadear uma atualização da interface para a refletir.
- Os *hooks* são utilizados para componentes poderem afetar estados e notificar o *React* dessas alterações. Na *framework* foram criados alguns *hooks* importantes para os componentes do desenvolvedor comunicarem com os componentes da *framework*.
- Os *contexts* servem para guardar informação que pode ser acedida, não só pelo componente, mas por toda a árvore descendente do mesmo. Na *framework* existe um *context* especifico para guardar os componentes de jogo criados pelo desenvolvedor.

Todos os componentes criados para mostrar informação ao utilizador, utilizam a biblioteca *Material UI*[7]. Esta biblioteca contem diversos componentes que já contêm a estilização criada, podendo esta ser alterada em cada instância para personalizar o elemento.

4.3.1 Componentes de Página

Como já foi referido, as páginas da aplicação têm todas um componente que vai servir como base para o resto da estrutura. Estes componentes não recebem qualquer tipo de propriedades, mas como foi mencionado na secção de pedidos, eles podem ter um *loader* auxiliar que faz um pedido ao servidor para obter a informação necessária.

Todas as páginas que o cliente pode aceder têm presente uma barra de navegação, que é um componente chamado **NavBar**. Nesta barra é possível ir para a página inicial e para as várias páginas relacionados com o utilizador e autenticação.

4.4 Autenticação e Cookies

Existem algumas páginas, presentes na *framework*, que necessitam de autenticação para poderem ser acedidas, como por exemplo, a página com a informação do próprio utilizador.

Para estas ocasiões foi criado o componente **RequireAuth**. Este componente utiliza a biblioteca *React Cookie*[15], para aceder ao *cookie "login"* que é criado quando o utilizador se autentica na aplicação.

Este *cookie*, quando é criado, consiste num booleano, *"loggedIn"*, que está com o valor lógico de *true*, e uma *String*, *"username"*, com o nome de utilizador que acabou de se autenticar. Quando o utilizador faz *logout*, o valor de *"loggedIn"* passa a *false* e o *"username"* é removido.

O *RequireAuth* é utilizado no lugar do componente base da página, sendo este passado como filho. Se um utilizador não autenticado tentar aceder a uma página que requer autenticação, o *RequireAuth* lança um alerta ao utilizador e em seguida redireciona-o para a página de *Login* onde o mesmo se pode autenticar.

4.5 Games Provider

O componente responsável por guardar todos os jogos implementados pelo desenvolvedor é o **Games Provider** que contem um mapa chave-valor, onde a chave é o nome do jogo e o valor é o componente *React* base criado pelo desenvolvedor. Esta informação é guardada num *context*, que é uma forma que o *React* tem para guardar informação que componentes filhos podem aceder, sem ter de a passar nas propriedades. Isto deixa o código mais organizado e fácil de ler.

4.6 Utilizar a Framework de Frontend

Para um desenvolvedor poder adicionar o seu jogo à aplicação, este necessita de ser adicionado ao mapa referido no último paragrafo. A nível do componente base que necessita ser criado, este precisa de implementar as propriedades do tipo *GameProps*. O *GameProps*

consiste no id do utilizador, na partida e em funções criadas para poder enviar os *setups* e *turns* que o jogador faça para o servidor.

A propriedade representante da partida, *match*, é atualizada automaticamente, devido ao *pooling* mencionado na secção de pedidos, que é feito sempre que é a vez do oponente de jogar. Isto garante que quando houver uma alteração na partida no servidor, essa informação é refletida no componente. Ambas as funções *doSetup()* e *doTurn()* têm uma *action* do tipo *any* nos seus argumentos, que é utilizado para passar a informação especifica da configuração e da jogada respetivamente. Cabe ao desenvolvedor utilizar, nos seus componentes, estas funções sempre que um utilizador realizar ou um *setup* ou um *turn*.

Idealmente o desenvolvedor, vai utilizar componentes disponibilizados pela biblioteca *Material UI*[7]. Isto vai permitir utilizar a estilização que já está definida na *framework*, para além de permitir fazer alterações a essa estilização no próprio componente.

Capítulo 5

Conclusão

Os objetivos do projeto foram concluídos com sucesso. As duas *frameworks* ficaram intuitivas e pouco restritivas para os programadores que as venham a utilizar e todas as funcionalidades implementadas foram testadas com sucesso.

Os jogos implementados para experimentar o projeto foram:

- O jogo do galo - Este jogo é simples e foi usado maioritariamente para testes.
- A batalha naval - Este jogo é relativamente complexo, mas conhecido pelo público.
- A corrida no castelo - Este jogo foi conceptualizado por um dos alunos e é a base de peças.

Todos os jogos criados são jogos de tabuleiro, mas também é possível utilizar as *frameworks* para criar um jogo de cartas baseado em turnos.

Este projeto foi criado a partir da vontade de ambos os alunos terem de criar um jogo para o projeto final, ao que os professores mencionaram a possibilidade de criar uma *framework* que tentasse generalizar as partes comuns a jogos baseados em turnos. As partes de generalização de todos os jogos de turno e conceptualização do produto final em ambas as *frameworks* foram o que se provou mais complicado de realizar no projeto inteiro, em contraste a implementação das interfaces, durante a criação dos jogos, foi simples e direta.

A escolha de tecnologias ter partido principalmente da familiaridade que ambos os alunos tinham provou ser uma escolha positiva, porque facilitou as revisões do código escrito e facilitou o objetivo principal que foi aprender a fazer *frameworks*, generalizar ideias diversas e a fazer código modular que vai ser utilizado por outros programadores. A utilização tanto do *Kotlin* como do *TypeScript* facilitou a criação das interfaces, uma vez que ambas as linguagens têm "tipos"e as propriedades de cada objeto, que é processado, já são conhecidas quando estes chegam a *framework*.

O *React* foi uma escolha que encaixou bem com a ideia do projeto de ser reativo a mudanças que é essencial em jogos online e o *Spring*, em geral, também foi uma escolha que

consideramos correta, porque simplificou a transação dos pedidos e a forma como a aplicação é iniciada.

A escolha de guardar os utilizadores apenas com um *username* e com uma password faz com que a *framework* seja mais acessível a programadores que estejam a fazer um projeto mais pessoal para partilhar com amigos por exemplo.

O objetivo principal do projeto era a criação da *framework* e sentimos que apesar de poderem ser adicionadas mais funcionalidades que poderiam ser interessantes, o objetivo foi concluído com sucesso. E em termos dos jogos, a lógica e o domínio ficaram completas e funcionais, podendo apenas ter sido melhorado o aspeto da interface de utilizador.

Bibliografia

[1] Kotlin. https://kotlinlang.org/. [Online; accessed 20-March-2023].

[2] Spring. https://spring.io/. [Online; accessed 20-March-2023].

[3] Postgresql. https://www.postgresql.org/. [Online; accessed 20-March-2023].

[4] Jdbi. https://jdbi.org/. [Online; accessed 20-March-2023].

[5] Typescript. https://www.typescriptlang.org/. [Online; accessed 20-March-2023].

[6] React. https://react.dev/. [Online; accessed 20-March-2023].

[7] Material-ui. https://mui.com/. [Online; accessed 20-March-2023].

[8] Siren hypermedia. https://github.com/kevinswiber/siren. [Online; accessed 03-June-2023].

[9] Problem details for http apis. https://datatracker.ietf.org/doc/html/rfc7807. [Online; accessed 03-June-2023].

[10] Json. https://www.json.org/json-en.html. [Online; accessed 04-September-2023].

[11] Jackson. https://github.com/FasterXML/jackson. [Online; accessed 04-September-2023].

[12] Single page application. https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application. [Online; accessed 04-September-2023].

[13] Fetch api. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. [Online; accessed 01-September-2023].

[14] Promisse. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Online; accessed 01-September-2023].

[15] React cookie. https://www.npmjs.com/package/react-cookie. [Online; accessed 04-September-2023].

Apêndice A

Apêndices

A.1 Uris

A.1.1 User

- /user/register

  `register(@RequestBody input: RegisterInputModel): ResponseEntity<*>`

  Este método processa um pedido POST para o registo de um novo utilizador. Chama o serviço **userServices.createUser** para criar um novo utilizador com base nas informações fornecidas. Retorna uma resposta, em caso de sucesso, com uma representação Siren.

- /user/login

  `login(@RequestBody input: LoginInputModel): ResponseEntity<*>`

  Este método processa um pedido POST para autenticar o utilizador e criar um token de login. Chama o serviço **userServices.createToken** para criar o token com base nas credenciais fornecidas. Em caso de sucesso, retorna uma resposta, com uma representação Siren vazia, contendo o token criado no header cookie, sendo este HTTP only.

- /user/logout

  `logout(user: User): ResponseEntity<*>`

  Este método processa um pedido POST para fazer logout do utilizador. Atualiza o status do utilizador para "offline"na base de dados. E envia um cookie com tempo de vida com 0 segundos. Isto faz com que o browser remova o cookie de autenticação.

- /user/me

```
getMe(user: User): ResponseEntity<*>
```

Este método processa um pedido GET para obter os detalhes do próprio utilizador. Chama o método getById passando o ID do utilizador atual.

- /user/id

```
getById(@PathVariable id: String): ResponseEntity<*>
```

Este método processa um pedido GET para obter os detalhes de um utilizador pelo seu ID. Chama o serviço **userServices.getUserById** para obter as informações do utilizador com base no ID fornecido. Retorna uma resposta com uma representação Siren dos detalhes do utilizador encontrado ou uma resposta de problema se o utilizador não for encontrado.

A.1.2 Game

- /gameList

```
getGameList(): ResponseEntity<*>
```

Este método processa um pedido GET para obter a lista de jogos disponíveis. Chama o serviço **gameServices.getGameList** para obter a lista de jogos. Retorna uma resposta com uma representação Siren da lista de jogos.

- /game/nameGame

```
getGameInfo(@PathVariable nameGame: String): ResponseEntity<*>
```

Este método processa um pedido GET para obter informações sobre um jogo específico. Chama o serviço **gameServices.getGameInfo** para obter as informações do jogo com base no nome no url do pedido. Retorna uma resposta com uma representação Siren das informações do jogo encontrado ou uma resposta de problema se o jogo não existir.

- /game/nameGame/find

```
findMatch(
    user: User,
    @PathVariable nameGame: String
): ResponseEntity<*>
```

Este método processa um pedido POST para procurar uma partida para o utilizador num jogo específico. Chama o serviço **gameServices.findMatch** para iniciar a pesquisa por uma partida. Em caso de sucesso retorna uma representação vazia Siren com redirecionamento para /game/nameGame/found.

- /game/nameGame/found

```
foundMatch(
    user: User,
    @PathVariable nameGame: String
): ResponseEntity<*>
```

Este método processa um pedido GET para verificar se uma partida foi encontrada para o utilizador em um jogo específico. Chama o serviço **gameServices.foundMatch** para verificar se uma partida foi encontrada. Retorna uma resposta com uma representação Siren da partida encontrada ou uma resposta de problema se a partida não existir ou se o utilizador não estiver nela.

- /game/nameGame/match/id

```
getMatchById(
    user: User,
    @PathVariable nameGame: String,
    @PathVariable id: String
): ResponseEntity<*>
```

Este método processa um pedido GET para obter os detalhes de uma partida pelo seu ID. Chama o serviço **gameServices.getMatchById** para obter as informações da partida com base no ID fornecido. Retorna uma resposta com uma representação Siren dos detalhes da partida encontrada ou uma resposta de problema se a partida não existir.

- /game/nameGame/setup

```
setup(
    user: User,
    @PathVariable nameGame: String,
    @RequestBody setup: SetupInputModel
): ResponseEntity<*>
```

Este método processa um pedido POST para configurar uma partida. Chama o serviço **gameServices.setup** para configurar a partida com base nas informações fornecidas. Retorna uma reposta com a partida atualizada, ou em caso de erro de *input*, uma mensagem do problema.

- /game/nameGame/turn

```
doTurn(
    user: User,
    @PathVariable nameGame: String,
    @RequestBody turn: TurnInputModel
): ResponseEntity<*>
```

Este método processa um pedido POST para realizar uma jogada em uma partida. Chama o serviço **gameServices.doTurn** para processar a jogada do utilizador na partida. Retorna uma reposta com a partida atualizada, ou em caso de erro de *input*, uma mensagem do problema.

- /game/nameGame/myturn

```
isMyTurn(
    user: User,
    @PathVariable nameGame: String,
    @PathVariable id: String
): ResponseEntity<*>
```

Este método processa um pedido GET para verificar se é a vez do utilizador em uma partida. Chama o serviço **gameServices.isMyTurn** para verificar se é a vez do utilizador na partida. Retorna uma resposta com o resultado da verificação.