Instituto Superior Engenharia de Lisboa

Área Departamental de Engenharia Informática

CrossBoard Application

Autores:
Luís Reis - 48318
Rúben Louro - 48926

Orientador:
Pedro Pereira

Projeto e Seminário

Semestre de Verão 2024/2025

02-06-2025

Resumo

No quotidiano atual, a tecnologia assume um papel cada vez mais central na realização das nossas tarefas diárias, recorrendo a diversos dispositivos, como smartphones, computadores, consolas de videojogos, microondas, impressoras, automóveis (através dos computadores de bordo), entre outros. O uso destas ferramentas tornou-se extremamente comum, destacando-se especialmente os smartphones e computadores, utilizados para consultar e-mails, comunicar com outras pessoas ou realizar atividades de lazer.

O desenvolvimento de aplicações tem sido profundamente influenciado por esta crescente utilização por parte dos consumidores finais, o que representa um dos principais desafios no panorama da programação atual.

Atualmente, a criação de aplicações tende a privilegiar abordagens multiplataforma, de modo a permitir a sua utilização em diferentes tipos de dispositivos. Desenvolver separadamente a mesma aplicação para plataformas distintas implica um aumento significativo dos custos, devido à duplicação de código, ao esforço adicional de desenvolvimento e à complexidade acrescida na manutenção.

Para enfrentar este desafio, recorresse cada vez mais, a linguagens e tecnologias que suportam o desenvolvimento multiplataforma. Será explorada a utilização do **Kotlin Multiplatform (KMP)** [1] , permitindo a partilha de código comum entre diferentes dispositivos (como computadores, smartphones, entre outros), mantendo ao mesmo tempo a possibilidade de escrever código específico para cada plataforma. Este modelo contribui para a redução da duplicação de código, do tempo de desenvolvimento e dos custos de manutenção.

Em suma, foi desenvolvida uma aplicação multiplataforma designada **CrossBoard Application**, utilizando Kotlin Multiplatform, no contexto de vídeo jogos de tabuleiro. Esta aplicação é compatível com smartphones, computadores e browsers, com o objetivo de demonstrar as potencialidades do desenvolvimento multiplataforma. Inicialmente, inclui pelo menos dois jogos, com a possibilidade de expansão para outros, sendo estruturada de forma genérica para permitir que outros desenvolvedores possam facilmente adicionar novos jogos.

Os principais objetivos deste projeto são: maximizar a reutilização de código entre o cliente e o servidor, bem como entre as diferentes plataformas, e facilitar tanto a adição de novos jogos como a manutenção da aplicação.

Abstract

In today's world, technology plays an increasingly central role in carrying out our daily tasks, relying on a wide range of devices such as smartphones, computers, video game consoles, microwaves, printers, and automobiles (through onboard computers), among others. The use of these tools has become extremely common, with smartphones and computers standing out in particular for checking emails, communicating with others, or engaging in leisure activities.

Application development has been deeply influenced by this growing usage among end consumers, which represents one of the main challenges in the current programming landscape.

Nowadays, application development tends to favour cross-platform approaches, enabling applications to run on various types of devices. Developing the same application separately for different platforms significantly increases costs, due to code duplication, additional development effort, and greater maintenance complexity.

To address this challenge, developers increasingly turn to languages and technologies that support cross-platform development. In this context, Kotlin Multiplatform will be explored, allowing the sharing of common code across different devices (such as computers, smartphones, and others), while also enabling the implementation of platform-specific code. This approach helps reduce code duplication, development time, and maintenance costs.

In summary, a cross-platform application named **CrossBoard Application** was developed using Kotlin Multiplatform (KMP), within the context of board videogame applications. This application is compatible with smartphones, computers, and web browsers, aiming to demonstrate the capabilities of cross-platform development. Initially, it will include at least two games, with the potential for expansion, and is designed in a generic way to allow other developers to easily integrate new games.

The main goals of this project are: to maximise code reuse between the client and server, as well as across different platforms, and to facilitate both the addition of new games and the maintenance of the application.

Índice

Índice de figuras

Índice de Listagens

1.Introdução

O projeto **CrossBoard Application** é uma aplicação que visa explorar as funcionalidades da tecnologia Kotlin Multiplatform, a qual tem como objetivo o desenvolvimento de aplicações multiplataforma, maximizando a partilha de código entre cliente e o servidor, e facilitando a manutenção.

No nosso quotidiano, é evidente a existência de muitas aplicações comuns entre diferentes plataformas, como o computador, smartphone ou browser. No entanto, ao serem desenvolvidas especificamente para uma determinada plataforma, a sua adaptação a outras requer, muitas vezes, o desenvolvimento de diferentes aplicações distintas, aumentando significativamente o custo e a complexidade do processo. A utilização de tecnologias como o Kotlin Multiplatform visa facilitar esse desenvolvimento.

O projeto **CrossBoard Application** consiste numa aplicação multiplataforma de jogos de tabuleiro (computador, smartphone, browser) demonstrando como a generalização e reutilização de código facilitará a adição de novas funcionalidades (neste caso de jogos). No caso desta aplicação, tal será exemplificado através da implementação de dois tipos diferentes de jogos de tabuleiro, sendo o TicTacToe [2] e o Reversi [3].

O utilizador pode realizar diversas ações na aplicação, sendo possível utilizar a aplicação sem registo, jogando em modo Singleplayer (Jogo contra o computador), ou, no caso opte por registar-se, tem acesso a funcionalidades adicionais, como jogar contra outros jogadores em diferentes plataformas.

Em suma, a aplicação **CrossBoard Application** pretende demonstrar o potencial do Kotlin Multiplatform, integrando diversas funcionalidades e jogos, os quais serão apresentados posteriormente. Neste capítulo, serão ainda apresentados a motivação para o desenvolvimento do projeto, os objetivos e especificações do mesmo, bem como a estrutura do relatório.

1.1 Motivação

Como referido anteriormente, o desenvolvimento de aplicações, atualmente,deve considerar a diversidade de acesso, como smartphones, computadores, browser, etc. A utilização de tecnologias como o Kotlin Multiplatform, permite que essas aplicações estejam disponíveis em várias plataformas, facilitando o seu desenvolvimento e manutenção.

O Kotlin Multiplatform destaca-se pelo seu principal objetivo: a reutilização máxima de código comum. Esta abordagem permite centralizar a lógica de negócio, lógica de dados, e

outras funcionalidades partilhadas numa única base de código partilhada (Shared). Para cada plataforma, apenas será necessário desenvolver um código específico, como interfaces gráficas ou integração de APIs nativas. A utilização da linguagem Kotlin, contribui também para a simplicidade e eficiência no desenvolvimento.

Entre as principais vantagens da adoção desta tecnologia, destacam-se:

- Reutilização de código entre diferentes plataformas
- Manutenção mais simples
- Redução de custos de desenvolvimento

Com os conhecimentos adquiridos ao longo do curso Licenciatura em Engenharia de Informática e de Computadores [4], no Instituto Superior de Engenharia de Lisboa [5], nomeadamente em unidades curriculares como Desenvolvimento de Aplicações Web (DAW), Técnicas de Desenvolvimento de Software (TDS), Laboratório de Software (LS) e Programação em Dispositivos Móveis (PDM), surgiu a ideia de desenvolver uma aplicação que integrasse esses conhecimentos.

Pretendemos que a aplicação utilize os conceitos abordados nestas disciplinas, tendo como objetivo principal a criação de uma solução para diferentes plataformas. Foi nesse contexto que nos foi apresentada a tecnologia Kotlin Multiplatform, despertando o nosso interesse tanto pela utilização da linguagem Kotlin como pelo seu propósito de permitir o desenvolvimento multiplataforma.

Concluímos, assim, pela utilização desta tecnologia no desenvolvimento da nossa aplicação, escolhendo o conceito de jogos de tabuleiro, o que nos permite explorar a tecnologia de forma mais prática e adaptada ao nosso contexto. Esta escolha possibilitou-nos, não só, o contacto com uma nova tecnologia de programação, como também o aperfeiçoamento de competências importantes como o trabalho em grupo, o planeamento e o desenvolvimento de projetos.

1.2 Objetivos e Especificações do Projeto

Para que a aplicação consiga alcançar os objetivos pretendidos, foram definidos os seguintes requisitos e funcionalidades:

- Os utilizadores podem registar-se na aplicação, embora o registo não seja obrigatório.
- Um utilizador não registado pode jogar em Singleplayer (contra o computador).
- Os utilizadores registados podem jogar em modo Singleplayer (contra o computador) ou Multiplayer (contra outro utilizador).

- Os utilizadores registados têm acesso a estatísticas dos jogos realizados e podem consultar os dados do seu perfil.
- Os utilizadores registados podem manter a sessão iniciada mesmo após fecharem a aplicação, caso assim o desejem.
- Os utilizadores podem iniciar um novo jogo após a conclusão de uma partida, se assim o pretenderem.
- Caso um utilizador desista de uma partida, será registada uma derrota e uma vitória para o adversário.
- Os administradores têm a capacidade de banir e desbanir utilizadores.
- Os administradores podem consultar informações de qualquer jogador.
- A aplicação está disponível em várias plataformas (Android, IOS, Desktop, browser, etc).
- A aplicação proporciona uma experiência de utilização intuitiva e amigável (user-friendly).
- A documentação da aplicação é completa, de forma a facilitar futuras atualizações por parte de outros desenvolvedores, seja para adicionar funcionalidades ou suportar novos jogos.
- A aplicação apresenta uma boa cobertura de testes nas diferentes funcionalidades, de modo a garantir segurança e confiança para o utilizador.

1.3 Método de Trabalho

O projeto foi desenvolvido com recurso à plataforma GitHub [6], utilizando um repositório privado para o efeito. Durante a realização do projeto, foram exploradas várias funcionalidades da plataforma, nomeadamente a secção **Code**, que utilizamos para armazenar e partilhar o código da aplicação, facilitando assim a colaboração e o acompanhamento do progresso entre os membros do grupo e o orientador.

Foi igualmente utilizada a funcionalidade **Issues**, com a criação de **Milestones** para assinalar objetivos semanais ou metas definidas entre diferentes fases do projeto. Também serve para registar problemas identificados na aplicação e acompanhar a respetiva resolução.

Recorreu-se ainda à aplicação *Docker [7]* para a criação do container do servidor da aplicação, cujo arquitetura será detalhada na secção Arquitetura do Server.

Adicionalmente, foi utilizada a aplicação *DBeaver [8]* para o desenvolvimento e gestão da base de dados da aplicação, a qual será igualmente descrita maioritariamente na mesma secção.

Esta abordagem, com o apoio da plataforma *GitHub*, permitiu um desenvolvimento paralelo mais eficiente, melhor organização de tarefas e prioridades, resolução colaborativa

de problemas, bem como uma comunicação mais eficaz entre os membros do grupo e o orientador.

1.4 Estrutura do Relatório

A estrutura do relatório está organizada em quatro capítulos.

No **Capítulo 1**, é apresentada uma introdução ao projeto, com uma explicação geral sobre o mesmo, incluindo as tecnologias e aplicações utilizadas.

No **Capítulo 2**, são identificados e descritos os principais aspetos da arquitetura da aplicação CrossBoard Application.

No **Capítulo 3**, são detalhados os aspetos relacionados com a implementação das diferentes secções do sistema da aplicação, bem como as motivações por detrás das decisões tomadas ao longo do desenvolvimento.

Por fim, no **Capítulo 4**, é apresentada a conclusão do trabalho realizado, abordando os desafios encontrados durante o desenvolvimento, uma reflexão crítica sobre o projeto e agradecimentos a todas as pessoas que contribuíram para a sua concretização.

2.Arquitetura

Neste capítulo é descrita a arquitetura geral do sistema, sendo esta ilustrada pela seguinte figura:

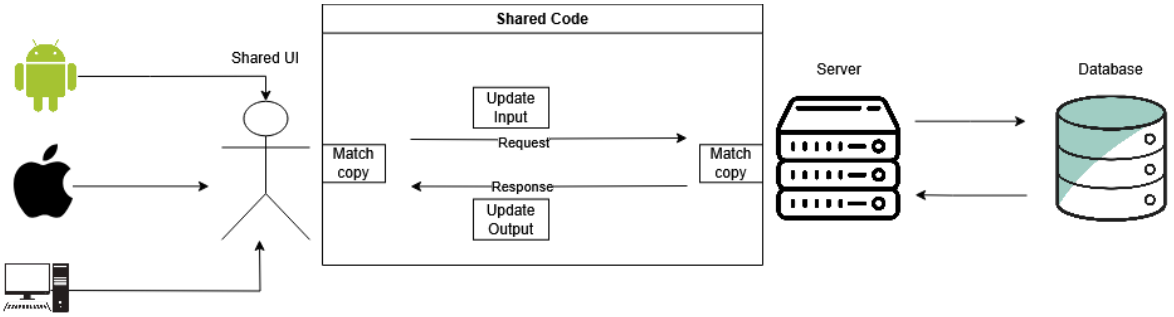

Figura 1 - Arquitetura geral do sistema

O sistema é dividido em três módulos, ***shared***, responsável por estruturar os dados utilizados no sistema e a interação entre os mesmos,que são utilizados tanto por servidor e cliente, e ainda código compartilhado entre plataformas mas com diferentes implementações, para cada um deles. A camada ***server***, responsável pela implementação de código específico de servidor, e a camada ***composeApp***, onde é implementado a grande maioria de código cliente e a sua interface gráfica (UI), que é partilhado entre todos os diferentes tipos de plataformas.

2.1 Shared

O módulo ***shared*** contém a **camada Domain**, que é responsável pela estruturação dos dados utilizados e a lógica implementada entre os dados. É também responsável por estruturar os dados incluídos no corpo de mensagens HTTP (Http Model), desta forma, tanto o cliente como o servidor têm acesso à estrutura do corpo nas mensagens. Quanto à estruturação de código cliente, mas com diferentes implementações, é usada para, por exemplo, obter os locais de armazenamento de dados de sessão, de cada plataforma.

2.2 Server

O módulo ***server*** contém a implementação e configuração do servidor, fazendo recurso à tecnologia ***Ktor [9]*** e dispõe uma Web API, que através de diversos *endpoints*, disponibiliza as funcionalidades necessárias para os clientes. Quando um *endpoint* é acedido, este faz recurso a camadas ***service*** que são responsáveis por obter, criar ou alterar estados de dados necessários de forma a concretizar a funcionalidade pretendida. Para o serviço

conseguir guardar estes dados, é necessária a implementação de uma camada ***Data Access***, que é responsável pela troca de dados entre o servidor e a base de dados, utilizando a tecnologia ***JDBC*** [10]. A base de dados é implementada sobre a tecnologia ***PostgreSQL*** [11] e mantida num *container*, disponibilizado pela tecnologia ***Docker***.

O esquema geral da Web API é ilustrado pela seguinte figure:

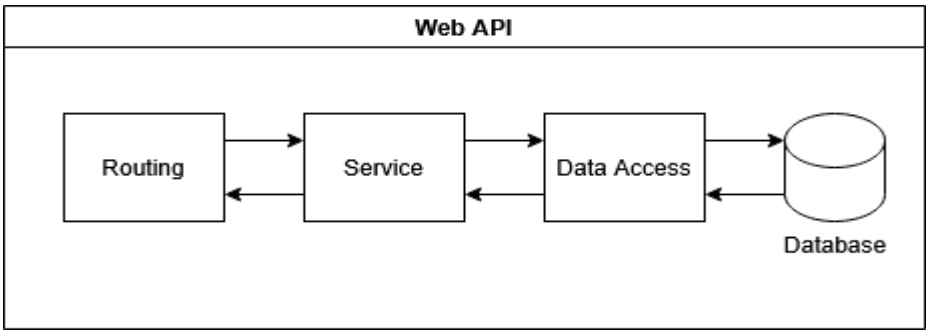

Figura 2 - Esquema geral da Web API.

2.3 ComposeApp

Neste módulo é implementada a UI, que através de ***Jetpack Compose*** [12] é permitida a partilha entre múltiplas plataformas. Para a implementação da UI, são utilizadas **Activities**, que são responsáveis por gerenciar os **ViewModels** e **Screens** necessários para cada estado de ecrã da aplicação. Cada ViewModel é responsável pelo armazenamento de estados de dados necessários para cada ecrã e a lógica utilizada para aplicar mudanças nestes dados. Os Screens são responsáveis pela representação visual de cada ecrã e a visualização de cada estado dos dados. É ainda implementada uma camada **API Client** , que fornece a lógica dos pedidos para cada *endpoint* fornecido pelo servidor. Esta camada é necessária para os ViewModels que através destes pedidos, conseguem realizar as mudanças de estados de dados, de acordo com a resposta dada pelo servidor.

As interações entre estas camadas são ilustradas na seguinte imagem:

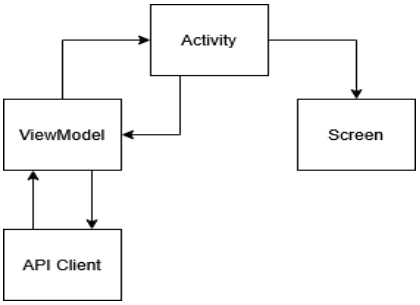

Figura 3 - Esquema de interações de atividades.

3.Implementação

Este capítulo é responsável por métodos de implementação das diferentes secções da aplicação.

3.1 Shared

Dentro do módulo ***shared***, existe uma diretoria "/commonMain", que detém código independente de qual a plataforma a ser utilizada e existem também diretorias que permitem código específico para uma plataforma ("/androidMain", "/iosMain", "jvmMain", etc.).

3.1.1 Domain

Para o desenvolvimento da aplicação são utilizados dados que respeitam o modelo de dados relacional na figura seguinte:

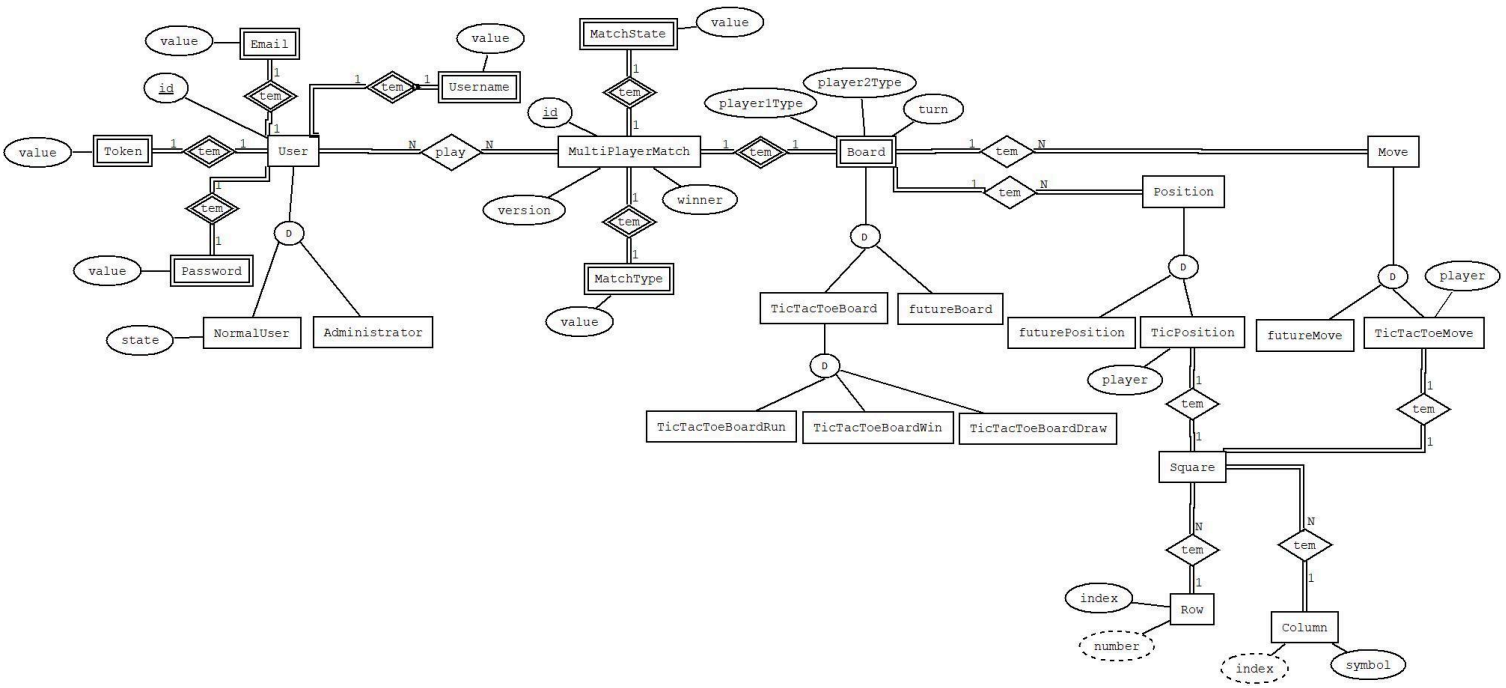

Figura 4 - Modelo de dados relacional.

Através deste modelo de dados temos um utilizador geral *User*, com um "Username", "Email", "Token" e "Password". O *user* poderá ser um utilizador normal *Normal User* ou um administrador *Admin*, em que ambos herdam as propriedades de *User*, mas o utilizador normal ainda tem uma propriedade extra "state", que pode ser NORMAL, para quando um utilizador tem um funcionamento normal na aplicação, ou BANNED, que restringe o acesso deste utilizador à aplicação.

Um utilizador pode jogar múltiplas partidas *MultiplayerMatch* contra outros utilizadores.

Uma partida, tem 2 jogadores, um "MatchState", que descreve o estado da partida, um "winner", que representa o vencedor da partida, um "MatchType", que representa o tipo de jogo a ser jogado (*TicTacToe, Reversi, etc.*), e um *Board*, que implementa a lógica do jogo a ser jogado.

Um *Board*, contém um "player1type" e um "player2type", que são o tipo de jogador atribuído a cada utilizador, e podem ser BLACK ou WHITE, contém ainda um "turn", que representa qual o jogador que a vez é para jogar, uma lista de jogadas *Move*, que representam cada jogada feita no jogo, e uma lista de posições *Position*, que representa o estado da posições do tabuleiro.

De forma a que seja possível implementar vários jogos "Turn based", é necessário que cada jogo tenha a sua implementação a respeitar o formato de *Board*, para que seja obtida a lógica do jogo pretendido, também é necessário respeitar o formato *Move*, visto que diferentes tipos de jogos podem ter diferentes tipos de jogadas a ser feitas, e ainda é necessário respeitar também o formato *Position*, porque os jogos também podem ter diferentes tipos de informação a guardar em cada posição, por exemplo, no caso de *TicTacToe*, é só necessário guardar a informação de qual o jogador que fez a jogada e qual o jogador que ocupa uma posição, mas outro tipo de jogos poderia ser necessário guardar, peças associadas a esse jogador.

Desta forma, é implementado um *TicTacToeBoard*, que respeita o formato *Board*, um *TicPosition*, que respeita o formato *Position*, contém um *Square* e o jogador nessa posição, e também um *TicTacToeMove*, que respeita o formato *Move*, e contém um *Square* e qual o jogador que fez a jogada. Para futuros tipos de jogos é necessário então estes criarem um novo "MatchType" associado a esse jogo, um *Board*, *Position* e *Move* para esse jogo.

Um *Square,* representa uma quadrícula do tabuleiro e contém um Row, que representa a linha e um Column, que representa a coluna. Um Row, contém índice "index" e um número "number". Um Colum contém um índice "index" e ainda um símbolo "symbol".

3.1.2 Http Model

Durante a comunicação entre o cliente e o servidor, por vezes é necessário enviar no corpo da mensagem HTTP, dados que precisam de respeitar um certo formato, sendo que o cliente envia formatos do tipo *Input* e o servidor do tipo *Output*. Através destes formatos, tanto o cliente como o servidor, conseguem saber o formato de dados que é esperado tanto no pedido como na resposta. De forma a esclarecer melhor esta situação, tem se o seguinte exemplo, para um utilizador que se quer registar na aplicação:
O utilizador "Bob" faz um registo de conta na aplicação, através de um pedido para o *endpoint* "/user" com o método POST. Para este pedido o formato do corpo terá de respeitar o seguinte formato input:

```
@Serializable
data class UserCreationInput(
    val username: String,
    val email: String,
    val password: String
)
```

Listagem 1 - Formato de dados para criação de utilizador.

Sabendo que tem de respeitar este formato, "Bob" envia os seguintes dados em formato JSON:

```
{
      "username": "Bob",
      "email": "bob@example.com",
      "password": "Aa1245!"
}
```

Listagem 2 - Exemplo de dados em JSON para criação de utilizador.

O servidor sabendo que o corpo respeita o formato pretendido, segue para criar um utilizador novo e se tudo correr com sucesso, envia os dados em falta para o utilizador construir um User com o seguinte formato:

```
@Serializable
data class UserCreationOutput(
     val id: Int,
     val token: String,
)
```

Listagem 3 - Formato de dados para dados gerados para um novo utilizador.

Ou então se algo correu mal, utiliza o seguinte formato:

```
@Serializable
data class ErrorMessage(
     val message: String
)
```

Listagem 4 - Formata para uma mensagem de erro.

Através do "status code" da resposta o utilizador consegue então perceber qual o tipo de formato vem na resposta.

3.1.3 Settings

Para guardar os dados de sessão num dispositivo, é necessário primeiro obter um local onde armazenar, mas para cada tipo de plataforma, o local de armazenamento de dados é diferente. Para tal, *Kotlin Mutiplatform* disponibiliza uma dialética, que permite dar forma a um método geral a ser utilizado utilizando as palavras chaves“expect fun” , mas com uma implementação específica em cada plataforma utilizando as palavras chaves “actual fun”. Desta forma, é criada então a seguinte função na diretoria “/commonMain”:

```
expect fun getSettings() : Settings
```

Listagem 5 - Esquema da função getSettings para cada plataforma

Nas diretorias específicas de plataforma, são então dadas as implementações de cada uma, por exemplo:

`"/androidMain":`

```
actual fun getSettings(): Settings {
    return
SharedPreferencesSettings(applicationContext.getSharedPreferences("setti
ngs", Context.MODE_PRIVATE))
}
```

Listagem 6 - Implementação de getSettings para android.

3.2 Server

No módulo server, é implementado código que pertence exclusivamente ao servidor e não é partilhado com mais nenhum módulo. Para a implementação do servidor, é feito recurso à tecnologia ***Ktor***. Para o servidor disponibilizar as funcionalidades necessárias para o funcionamento da aplicação, é usada uma *Web API*, composta por 3 principais divisões, sendo estas, Routing, Service e Repository (Data Access).

3.2.1 Routing

Função

Nesta camada são implementados todos os endpoints disponibilizados pelo servidor. Cada endpoint pode ser acedido através de um caminho e um método HTTP e este, de seguida, analisa os dados fornecidos no "body", "query" ou "path" do pedido. Após validação dos dados recebidos, a camada service é encarregada de gerir os dados da maneira necessária, de modo a que a funcionalidade seja implementada. O endpoint fica à espera da conclusão do serviço e conforme se o serviço obteve sucesso ou insucesso a realizar a funcionalidade, é enviada uma resposta com um status code que reflete o resultado obtido e um corpo com dados relevantes ou uma mensagem de erro.

Documentação OpenAPI

Uma descrição OpenAPI [13] é um formato utilizado para descrição formal de Web API's. Para a produção automática deste tipo de documentação é utilizada uma biblioteca [14] que através da configuração de um plugin no servidor Ktor, permite adicionar aos endpoints, informação extra que será usada na criação do documento.

Autenticação

Algumas funcionalidades são restritas a utilizadores que já estejam registados na aplicação. Para verificar que o pedido é feito por alguém registado na aplicação, é verificado o *Header Authorization.* Este campo precisa obrigatoriamente de conter um *Token,* que é associado a apenas um único utilizador, e ainda respeitar o formato "Bearer <token>".

Corpo das mensagens HTTP

Os dados contidos dentro do corpo das mensagens HTTP, respeitam o formato JSON. Este formato foi escolhido, devido à familiaridade desenvolvida com este ao longo de diversas cadeiras realizadas no curso.

Na camada ***shared*** são desenvolvidos diferentes tipos de formatos de dados (HTTP Model), que têm de ser respeitados, para cada troca de mensagens HTTP, de forma a que tanto o cliente como o servidor saibam que tipo de informação é necessária ser contida no corpo da mensagem. Desta forma, o servidor, que recebe dados em formato JSON, consegue desserializar esses dados para os formatos que são esperados. De seguida, estes dados são convertidos em objetos do ***Domain***, para que os dados respeitem as regras esperadas para o correto funcionamento das funcionalidades. Caso os dados recebidos falhem em algum destes passos, uma resposta de erro será enviada.

Funcionalidades

Seguindo os métodos especificados anteriormente, são então disponibilizadas as seguintes funcionalidades:

- Criar um utilizador;
- Modificar um utilizador;
- Login;
- Obter os dados de um utilizador por token;
- Banir um utilizador (exclusivo para administradores);
- Desbanir um utilizador (exclusivo para administradores);
- Procurar uma partida;
- Cancelar a procura de uma partida;
- Desistir de uma partida;
- Obter a informação de uma partida;
- Realizar jogadas numa partida;
- Obter estatísticas de partidas de um utilizador.

Os detalhes dos diversos endpoints podem ser consultados através do ficheiro *OpenAPI.yaml* [15], disponibilizado no repositório *Github* do projeto.

3.2.2 Service

O serviço necessita de receber dados que respeitem as regras do ***Domain***, e verificar se com esses dados, consegue criar novos recursos, obter recursos existentes ou então alterar recursos, respeitando as regras do ***Domain***. Estes recursos são armazenados por uma base de dados *PostgreSQL* e para existir troca de recursos entre a base de dados e o serviço, é necessário uma camada intermediária Repository (Data Access).

Quando é verificada a consistência dos recursos, é retornado o recurso pretendido, mas em caso de erro, é retornado um objeto do tipo enumerado *ApiError,* que descreve o erro observado. Para que seja possível o retorno de duas estruturas de dados diferentes, é utilizada a ferramenta *Either*, disponibilizada pelos professores da cadeira de Desenvolvimento de Ambientes Web (DAW), quando realizamos esta cadeira.

3.2.3 Repository

O repositório é encarregado por obter recursos existentes na base de dados ou por realizar alterações a esta. Para realizar a conexão com a base de dados, é utilizada a tecnologia *JDBC* que permite a realização de transações de dados, com uma base de dados *PostgreSQL*, fazendo recurso a comandos *SQL*, que têm acesso à propriedades que caracterizam os recursos e realizam então as modificações na base de dados. O repositório disponibiliza métodos simples, como obter, criar, modificar ou alterar um recurso, e cabe ao serviço saber usar métodos necessários para realizar uma funcionalidade

3.2.4 Database

A base de dados é mantida dentro de um container, disponibilizado pela tecnologia *Docker*. Sobre este container é corrido um pequeno "script" *SQL*, para criação das tabelas necessárias para armazenar os diferentes tipos de recursos.

O script [16] pode ser consultado no repositório Github do projeto.

3.2 ComposeApp

Nesta camada, é implementado todo o código relacionado com o cliente. Esta camada é dividida por diretorias, sendo a principal diretoria "/commonMain", onde é contido todo o código partilhado entre as diversas plataformas. As restantes diretorias, são destinadas a código específico de cada plataforma. Este código específico, é referente às diferentes maneiras de lançamento da aplicação para cada plataforma e obtenção da correta implementação de métodos específicos da plataforma.

O código partilhado, consiste em todo o UI, que através de Jetpack Compose, é suportado pelas diferentes plataformas, e um API Client, que é responsável por estruturar e enviar os pedidos HTTP, necessários para os diversos endpoints da Web API, disponibilizada pelo servidor.

3.2.1 UI

A interface gráfica, respeita uma organização composta por Activity, ViewModel e Screen, organização este, descrita anteriormente pela figura 3.

Activity

Uma atividade, é responsável por gerir, um dos estados principais da aplicação cliente (Autenticação, Menu Principal, Partida, etc.). Esta atividade realiza a conexão entre estados de recursos disponibilizados por um ViewModel e os seus métodos, e encaminha este recurso e métodos para o Screen indicado para aquele estado da aplicação e recurso.

ViewModel

Cada ViewModel, é responsável por gerir os recursos necessários para um dado estado da aplicação cliente, utilizando a tecnologia Flow. Também disponibiliza um conjunto de métodos que permitem a modificação de um recurso, sendo que, estes métodos, necessitam de também utilizar os diversos pedidos disponibilizados pelo API Client, para que as alterações de recursos sejam confirmadas pelo servidor.

Flow

Flow [17] é uma tecnologia que permite gerir sequências de estados de um determinado recurso de forma assíncrona e sem bloquear a Thread principal, resultando numa elevada eficiência na obtenção de novos estados.

Screen

Um ecrã é responsável pela representação visual do estado da aplicação e de recursos referentes a esse estado da aplicação. Também é responsável por garantir que as interações do utilizador com certos elementos gráficos, provocam alterações a um dado recurso através de métodos disponibilizados por um ViewModel.

3.2.2 API Client

Esta camada estrutura os pedidos HTTP para os diversos endpoints da Web API, garantido que os caminhos dos endpoints são passados corretamente, os métodos do pedido são os corretos. Garante também que um pedido, se for necessário, recebe o Header Authorization. No corpo dos pedidos, é então inserido o formato esperado dos dados (HTTP Model) e depois convertido em JSON, com o Header ContentType a receber o valor "application/json", para indicar o formato do pedido. Quando a resposta é recebida, através do "status code", são obtidos os dados no formato Output, em caso de sucesso, ou então ErrorMessage, que descreve o erro obtido. É utilizado Either, para que seja possível retornar recursos diferentes conforme a resposta.

4. Conclusão

Na realização deste projeto, desenvolvemos novas capacidades e adquirimos conhecimentos que nos permitiram encontrar soluções para os diferentes desafios que surgiram ao longo do desenvolvimento da aplicação **CrossBoard Application.**

Inicialmente, surgiram dúvidas relativamente à utilização e organização da estrutura do sistema, uma vez que recorremos a uma tecnologia nova para nós, sendo o Kotlin Multiplatform. Após investigação e análise da documentação, bem como de alguns exemplos práticos, conseguimos compreender melhor a estrutura, o que nos orientou eficazmente na implementação da aplicação.

Durante o desenvolvimento, deparámo-nos com requisitos de diferentes níveis de complexidade, o que nos obrigou a procurar soluções mais específicas para os problemas

encontrados. Como já tínhamos tido contacto prévio com outras linguagens, como o Postgresql SQL, Kotlin e Kotlin Compose, em disciplinas anteriores do curso, o desenvolvimento da base de dados, do frontend e mesmo do backend foi facilitado. Este conhecimento prévio revelou-se uma vantagem, permitindo uma gestão de tempo mais eficiente e libertando tempo para aprofundarmos o estudo da tecnologia Kotlin Multiplatform.

Com o planeamento semanal orientado pelo nosso orientador, fomos gradualmente encontrando e implementando soluções para os vários requisitos da aplicação, o que tornou o processo de desenvolvimento mais fluido. Apesar de alguns requisitos mais complexos terem exigido mais tempo, a familiaridade com a integração entre o backend e frontend, adquirida anteriormente, foi fundamental para ultrapassar esses obstáculos.

Em suma, conseguimos alcançar os objetivos propostos para este projeto. Aprendemos a utilizar uma nova tecnologia e a explorar as suas funcionalidades, bem como a organizar e planear um projeto de maior envergadura. O resultado final é uma aplicação flexível, que facilita a implementação de novos jogos de tabuleiro por parte de outros desenvolvedores, e segurança, graças aos testes realizados às funcionalidades existentes.

4.1 Críticas Construtivas

No âmbito do nosso projeto, reconhecemos que poderíamos ter introduzido algumas modificações e melhorias na aplicação. No entanto, devido à necessidade de cumprir os prazos estabelecidos e à aprendizagem básica da tecnologia Kotlin Multiplatform, nem todas as ideias puderam ser concretizadas. Ainda assim, identificámos algumas melhorias que poderiam ser implementadas numa versão futura da aplicação:

- Adição de novas funcionalidades, como diferentes níveis de dificuldade nos jogos em modo Singleplayer e um sistema de ranking.
- Otimização da aplicação, tanto do lado do servidor como do cliente.
- Inclusão de novos jogos à aplicação, por exemplo a batalha naval.
- Substituição do mecanismo de polling utilizado para obter o novo estado das partidas por uma solução baseada em WebSockets.

4.2 Agradecimentos

No desenvolvimento do nosso projeto, queremos expressar o nosso agradecimento ao nosso orientador, Engenheiro Pedro Pereira, pela oportunidade de realizarmos um projeto sobre um tema do nosso interesse. Agradecemos também por nos ter apresentado a tecnologia Kotlin Multiplatform, que contribui para uma experiência mais enriquecedora ao longo do projeto e nos permitiu adquirir um maior conhecimento sobre tecnologias aplicáveis ao desenvolvimento de aplicações desta natureza. A sua disponibilidade, exigência e compromisso foram fundamentais para o sucesso deste trabalho, e por isso, deixamos o nosso sincero agradecimento.

Estendemos também o nosso reconhecimento ao Engenheiro Fernando Sousa, pelo seu excelente trabalho e dedicação na disciplina de Projeto e Seminário, proporcionando-nos uma experiência positiva, bem como uma organização exemplar da unidade curricular.

Por fim, agradecemos ao Instituto Superior de Engenharia de Lisboa pela oportunidade e pelos recursos disponibilizados ao longo da disciplina de Projeto e Seminário. Agradecemos igualmente a todos os docentes que nos acompanharam durante o percurso académico, contribuindo para um ambiente de aprendizagem estimulante e para o desenvolvimento das nossas competências na área da Engenharia Informática, preparando-nos de forma sólida para os desafios da vida profissional.

Referências

[1] https://kotlinlang.org/docs/multiplatform.html#, Kotlin Multiplatform.
[2] https://pt.wikipedia.org/wiki/Jogo_da_velha, TicTacToe.
[3] https://pt.wikipedia.org/wiki/Reversi, Reversi.
[4] https://www.isel.pt/, Instituto Superior de Engenharias de Lisboa (ISEL).
[5]https://www.isel.pt/curso/licenciatura/licenciatura-em-engenharia-informatica-e-de-computadores, ISEL, Licenciatura de Engenharia informática e de computadores (LEIC)
[6] https://github.com/, Github.
[7] https://www.docker.com/, Docker.
[8] https://dbeaver.io/, Dbeaver.
[9] https://ktor.io/, Ktor.
[10] https://en.wikipedia.org/wiki/Java_Database_Connectivity, Java Database Connectivity.
[11] https://www.postgresql.org/, PostgreSQL.
[12] https://developer.android.com/compose, Jetpack Compose.
[13] https://www.openapis.org/, OpenAPI.
[14] https://github.com/SMILEY4/ktor-openapi-tools, Github, SMILEY4, ktor-openapi-tools.
[15] https://github.com/PS-Grupo24/CrossBoard-Application/blob/main/docs/OpenAPI.yaml, Documentação OpenAPI do servidor do Projeto CrossBoard Application.
[16]https://github.com/PS-Grupo24/CrossBoard-Application/blob/main/code/CrossBoard%20Application/server/src/main/kotlin/com/crossBoard/sql/createTable.sql, Script SQL de geração de tabelas do projeto CrossBoard Application.
[17] https://kotlinlang.org/docs/flow.html#flows, Kotlin Flow.