Instituto Superior de Engenharia de Lisboa (ISEL)
Departamento de Engenharia Eletronica e de Telecomunicacoes e Computadores (DEETC)

LEIM
Licenciatura em Engenharia Informática e Multimédia
Unidade Curricular de Projeto

# Nostalgia Hunting


Rui Lança (42359)

Pedro Gomes (43765)

*Orientadora*

*Professora* Ana Correia

*Julho, 2025*

# Resumo

O Projeto consiste na criação de uma 'App' para Android utilizando como linguagem de programação Kotlin e uma API em NodeJs, utilizando as tecnologias "Graphql"e "TypeScript".

O Objetivo da aplicação é simular o processo de "tracking"de jogos e proezas criados por e para jogadores em diversos videojogos e para os mesmos jogadores poderem criar desafios, onde competem uns com os outros.

Para tal efeito foi realizado um sistema para utilizadores poderem criar, adicionar e terminar proezas relacionadas a vários jogos.

O Utilizador tem a liberdade de explorar a aplicação criando e terminando proezas em jogos que quiser competir.

A motivação para este projeto foi em realizar uma aplicação que permitisse jogar estes videojogos antigos de uma forma diferente, criando desafios diferentes dos que os desenvolvedores teriam inicialmente pensado.

# Abstract

This project consists of creating an Android application using the Kotlin programming language and a Node.js API, integrating GraphQL and TypeScript technologies.

The main goal of the app is to simulate a tracking system for games and achievements created by and for players across various video games. It also allows users to create custom challenges where multiple players can compete with each other.

To achieve this, a system was developed that enables users to create, add, and unlock achievements related to different games.

Users are free to explore the app by creating and completing achievements in any games they choose to engage with.

The motivation behind this project was to develop an application that allows players to enjoy classic video games in a new way, by introducing custom challenges beyond what the original developers had imagined.

# Agradecimentos

Gostaria de agradecer a Engenheira Ana Correia não so pela ajuda durante o projeto, mas também pela disponibilidade para retirar dúvidas e reforçar certos aspetos do projeto. Um agradecimento especial ao Engenheiro Pedro Fazenda por ter sido o nosso coordenador inicial para este projeto e por nos dar um conjunto de tecnologias e referências iniciais para o começo deste projeto. Aproveito também para agradecer aos professores Hélder Bastos e Paulo Trigo por lecionarem a cadeira de Projeto e disponibilizado um conjunto de ferramentas aos alunos para auxílio do projeto. Gostaria também de agradecer aos meus amigos e família por me apoiarem durante este processo académico.

Pedro Gomes

Gostaria de agradecer ao professor Pedro Fazenda por nos ajudar no ano passado a pesquisar as tecnologias utilizadas neste trabalho e na estrutura do trabalho, bem como o reforço dado pela professora Ana Correia. Gostaria também de agradecer aos meus pais pela paciência que eles tiveram com o tempo que eu demorei a terminar o curso.

Rui Lança

*Eventual texto de dedicatória ...*

*... mais texto,*

*... e o fim do texto.*

# Índice

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1

# Introdução

Atualmente, os videojogos fazem parte da vida de muitas pessoas, estando presentes desde 1951, ano em que começaram a ser desenvolvidos os primeiros protótipos de videojogos, até aos dias de hoje, em que são lançados milhares de jogos anualmente. Inicialmente, os videojogos tinham um estilo simples, sem muitas funcionalidades. Desde o lançamento de Pong em 1972 até aos anos 80 e 90 com os lançamentos de Pac-Man (1980) e Chrono Trigger (1995), é possível observar uma grande evolução. Estes jogos são bastante distintos entre si, tanto em termos tecnológicos como funcionais, mas existe algo que os une: o facto de serem considerados jogos do estilo retro.

O termo retro é utilizado para descrever algo antigo ou que transmite um sentimento de nostalgia. Embora a indústria dos videojogos tenha evoluído bastante, continua a existir uma grande parte da população que valoriza esse sentimento único associado a jogar algo antigo. Para esse efeito, foram criados vários emuladores com o objetivo de replicar consolas ou sistemas de videojogos antigos que já não são utilizados ou suportados pelos sistemas atuais.

Essa capacidade de jogar videojogos antigos em sistemas modernos abre portas a muitas possibilidades, como implementar funcionalidades normalmente associadas a jogos modernos nesses mesmos jogos antigos, transformando assim a experiência num conceito renovado.

Uma dessas funcionalidades, que pessoalmente mais me interessa, é a capacidade de obter proezas ao completar desafios em determinados jogos. Atualmente, alguns emuladores já suportam sistemas desse tipo ou similares, mas não incluem o aspeto social de completar desafios em conjunto com

outros jogadores. Essa interação social, permitindo competir uns contra os outros, tornaria a experiência de completar desafios mais gratificante e estimulante.

Este projeto consiste no desenvolvimento de uma aplicação (APP) em Kotlin e de uma API em NodeJS e GraphQL que suportem a gestão de vários "caçadores de proezas", onde os utilizadores podem não só adquirir proezas e competir com outros jogadores, mas também criar os seus próprios desafios (proezas), aumentando assim as possibilidades dos desafios para praticamente tudo o que os utilizadores considerarem interessante.

O utilizador pode consultar outros utilizadores, incluindo os desafios em que esses jogadores competem atualmente e as suas proezas obtidas. Todos os utilizadores têm também um ranking geral, onde é somado o valor de todas as suas proezas desbloqueadas. O utilizador pode igualmente desbloquear ou bloquear as suas próprias proezas, dependendo se consegue ou não completar o desafio, assim como criar as suas próprias proezas e jogos onde considerar interessante competir.

Os objetivos propostos para este projeto passam pelo desenvolvimento de uma APP que possa ser utilizada na maioria dos dispositivos Android. Para tal, foi necessário criar uma API capaz de suportar todas as funcionalidades pretendidas.

Para validar e testar o projeto, foram criados vários utilizadores (mocks), permitindo ao utilizador interagir livremente com o sistema.

O relatório segue a seguinte estrutura:

- Trabalho Relacionado (Capítulo 2): Neste capítulo são referenciados os vários videojogos e aplicações que foram utilizados como referência durante o desenvolvimento do projeto.

- Modelo Proposto (Capítulo 3): Este capítulo está dividido em subpontos nos quais são abordados os requisitos, os fundamentos, a abordagem, as características da APP e da API, a arquitetura e a interface do utilizador.

- Implementação do Modelo (Capítulo 4): Neste capítulo é descrito o modo de implementação e funcionamento do modelo proposto. São também apresentadas as dependências tecnológicas utilizadas.

- Validação e Testes (Capítulo 5): Neste capítulo são apresentados os testes realizados e os seus resultados após a finalização do projeto.
- Conclusões e Trabalho Futuro (Capítulo 6): Neste capítulo são apresentadas as conclusões do projeto e o trabalho futuro a desenvolver.

# Capítulo 2

# Trabalho Relacionado

Em termos de inspirações utilizadas ao longo do trabalho, o [Yellow, 1998] e o [Ratchet e Clank, 2004] foram 2 dos videojogos mais pensados para se utilizar a nossa aplicação. São 2 jogos relativamente fáceis de realizar desafios incluindo nós pessoalmente o termos feito no passado, embora sem qualquer forma de aspecto social, daí a ideia da criação da nossa aplicação. Não podíamos também acabar esta seleção sem nomear a importância do [Achievements, 2012] na criação da nossa aplicação. Esta é uma aplicação de computador open source ligada a um Emulador [Arch, 2012] que permite conquistar proezas em videojogos em 'real-time'. Esta aplicação teve um enorme peso na implementação do nosso projeto, pois inicialmente a nossa aplicação poderia ser pensada como uma extensão a essa aplicação na adição da componente social até decidirmos em realizar a nossa aplicação de forma completamente independente do [Achievements, 2012]. Houve também diversos jogos que serviram igualmente como inspiração, embora não tenham sido tão focados em. Alguns deles incluem:

- [Adventures, 2004]
- [Cooper e the Thievius Raccoonus, 2004]

# Capítulo 3

# Modelo Proposto

Ao longo deste capítulo será apresentado o modelo proposto, para que desta forma seja possível entender quais os requisitos, fundamentos, abordagens escolhidas e planeamento feitos ao longo do projeto.

## 3.1 Requisitos

Para que a análise de requisitos fosse feita da melhor forma e mais clara possível, foi realizada uma subdivisão desta secção: .

- A análise geral do problema, onde será analisada a síntese dos objetivos, público alvo e as metas a alcançar;
- A análise pormenorizada onde serão analisadas as funções e atributos do sistema, que surgem como resultado da análise geral;
- A análise dos casos de utilização, baseado nas necessidades e propósitos do sistema.

### 3.1.1 Análise Geral

Numa fase inicial foi necessário definir quais os requisitos a que o projeto devia corresponder, para que desta forma conseguíssemos criar o melhor modelo possível. Como tal, foi criada uma síntese de objetivos principais, o público-alvo para o projeto, bem como as principais metas pretendidas para o desenvolvimento do projeto.

Criação de uma 'APP' para telemóvel com o sistema operativo Android utilizando a linguagem de programação Kotlin. Criação também de uma API para complementar as funcionalidades da 'APP' utilizando a linguagem de Programação NodeJs com TypeScript e utilizando a tecnologia Graphql.

Todos os jogadores que tenham interesse em completar desafios e competir com outros utilizadores para testar quem termina os desafios mais rápido. A Aplicação é destinada a todas as faixas etárias.

Com esta aplicação pretende-se:

- Que os utilizadores tenham uma boa experiência enquanto exploram a 'APP'.
- Que a APP tenha um layout compreensivo e apelativo.
- Criar um ambiente divertido e competitivo.

### 3.1.2 Análise Pormenorizada

Numa fase mais avançada foi necessário indicar quais as funções e atributos do sistema, ou seja, as funções que é suposto realizar e que características o projeto deve ter.

#### Atributos do Sistema

Os atributos do sistema são características ou dimensões do sistema, alguns exemplos são: facilidade de utilização (ease of use), interação homem máquina (interface metaphor), entre outros. Estes podem variar num conjunto de valores de detalhe (attribute details), estes são normalmente valores discretos e simbólicos, juntamente estes atributos podem ter restrições de fronteira (attribute boundary constraints), que equivale usualmente a um intervalo de valores numéricos. Por fim, pode-se definir duas categorias para os atributos do sistema, Obrigatório ou Desejável. A tabela seguinte demonstra os atributos do nosso sistema:

| Atributo (Identificação do atributo) | Detalhe / Restrição Fronteira (Identificação dos detalhes e/ou valores do atributo) | Categoria (Obrigatório, Desejável) |
|---|---|---|
| interação pessoa-máquina | utilização do telemóvel (em vez do rato) | Desejável |
| | utilização do rato | Obrigatório |
| plataformas da aplicação | android studio, graphQL, MySQL, Firebase | Obrigatório |
| tempo de resposta ∈ {apropriado} | execução de funções rápida (removendo latência entre páginas) | Desejável |
| facilidade de utilização | pesquisas e navegação da aplicação rápida e intuitiva | Obrigatório |

Figura 3.1: Atributos do Sistema

### Requisitos do Sistema

Os requisitos são as descrições das necessidades do sistema, estas estão categorizadas em 3 tipos:

- Evidente - Tem que ser realizada. O utilizador tem que ter conhecimento da sua realização.
- Invisível - Tem que ser realizada. Não é visível para os utilizadores.
- Adorno - Opcional. Não afeta significativamente o custo ou outras funções.

As funções do sistema podem ser agrupadas, de modo a facilitar o processo (utilizadas são, Funções básicas (App), Funções de query da API e Funções de Firebase). A seguir pode-se observar estas funções, bem como as suas categorias nas tabelas seguintes.

| Requisito | Função - O sistema tem de fazer ... (Descrição resumida do requisito) | Categoria (Visível, Invisível, Adorno) | Agrupamento (a definir) |
|---|---|---|---|
| R1.1 | Criar uma conta de utilizador | Visivel | App |
| R1.2 | Efetuar log-in na aplicação | Visivel | App |

Figura 3.2: Requisitos do Sistema

| | | | |
|---|---|---|---|
| R1.3 | Modificar a conta de utilizador (nome e imagem) | Visivel | App |
| R1.4 | Pesquisar os jogos da biblioteca pessoal | Visivel | App |
| R1.5 | Adicionar jogos à biblioteca pessoal | Visivel | App |
| R1.6 | Remover jogos da biblioteca pessoal | Visivel | App |
| R1.7 | Pesquisar todos jogos da aplicação (incluindo por diferentes categorias) | Visivel | App |
| R1.8 | Adicionar jogos à aplicação | Visivel | App |
| R1.9 | Remover jogos da aplicação | Visivel | App |
| R1.10 | Editar os jogos da aplicação | Visivel | App |
| R1.11 | Pesquisar as conquistas de cada jogo | Visivel | App |
| R1.12 | Adicionar conquistas à aplicação | Visivel | App |
| R1.13 | Remover conquistas da aplicação | Visivel | App |
| R1.14 | Editar as conquistas da aplicação | Visivel | App |
| R1.15 | Terminar uma conquista | Visivel | App |
| R1.16 | Recomeçar uma conquista | Visivel | App |
| R1.17 | Verificar o ranking dos utilizadores | Visivel | App |
| R1.19 | Pesquisar outros utilizadores | Visivel | App |
| R1.20 | Navegação da aplicação | Visivel | App |
| R1.21 | Realizar pedidos de amizade a um utilizador | Visivel | App |
| R1.22 | Pesquisar os pedidos realizados pertinentes a um utilizador | Visivel | App |
| R1.23 | Aceitar os pedidos de amizade de utilizadores | Visivel | App |
| R1.24 | Rejeitar os pedidos de amizade de utilizadores | Visivel | App |
| R1.25 | Criar desafios | Visivel | App |
| R1.26 | Eliminar desafios | Visivel | App |
| R1.27 | Editar desafios | Visivel | App |
| R1.28 | Pesquisar os grupos a qual o utilizador pertence | Visivel | App |
| R1.29 | Pesquisar os grupos de utilizadores disponíveis (o utilizador não pertence) | Visivel | App |
| R1.30 | Pesquisar os vários desafios de um grupo | Visivel | App |
| R1.31 | Pesquisar todos os utilizadores pertencentes a um grupo | Visivel | App |
| R1.32 | Pesquisar todos os utilizadores pertencentes a um desafio | Visivel | App |
| R1.33 | Pesquisar todas as conquistas pertencentes a um desafio | Visivel | App |
| R1.34 | Verificar o progresso de todos os utilizadores num desafio | Visivel | App |
| R1.35 | Adicionar conquistas a um desafio aberto | Visivel | App |
| R1.36 | Remover conquistas a um desafio aberto | Visivel | App |
| R1.37 | Juntar-se a um desafio aberto | Visivel | App |
| R1.38 | Sair de um desafio | Visivel | App |
| R1.39 | Começar um desafio (fechar o desafio) | Visivel | App |
| R1.40 | Terminar uma conquista num desafio ao qual o utilizador pertença (desafio fechado) | Visivel | App |
| R1.41 | Terminar um desafio (quando houver vencedor) | Visivel | App |

Figura 3.3: Requisitos da API

| | | | |
|---|---|---|---|
| R1.42 | Registar as mudanças aos utilizadores da firebase (base de dados) | Invisivel | BD |
| R1.43 | Registar a informação dos jogos na biblioteca pessoal do utilizador (base de dados) | Invisivel | BD |
| R1.44 | Registar as mudanças aos jogos da aplicação (base de dados) | Invisivel | BD |
| R1.45 | Registar as mudanças às conquistas na aplicação (base de dados) | Invisivel | BD |
| R1.46 | Registar as mudanças às conquistas na biblioteca pessoal do utilizador (base de dados) | Invisivel | BD |
| R1.47 | Registar as mudanças aos utilizadores da aplicação (base de dados) | Invisivel | BD |
| R1.48 | Registar as mudanças aos pedidos de junção a listas de amigos (base de dados) | Invisivel | BD |
| R1.49 | Registar as mudanças aos pedidos de junção a desafios (base de dados) | Invisivel | BD |
| R1.50 | Registar as mudanças a desafios (base de dados) | Invisivel | BD |
| R1.51 | Registar as mudanças às conquistas no desafio (base de dados) | Invisivel | BD |
| R1.52 | Registar as mudanças aos utilizadores no grupo (base de dados) | Invisivel | BD |
| R1.53 | Registar as mudanças aos utilizadores no desafio (base de dados) | Invisivel | BD |
| R1.54 | Registar as mudanças à ligação entre os utilizadores e as conquistas nos desafios (base de dados) | Invisivel | BD |

Figura 3.4: Requisitos da Base de Dados

### 3.1.3 Casos de Utilização

Para os casos de utilização decidimos fazê-los com base nos vários requisitos que tínhamos no ponto anterior. Por isso estes casos de utilização serão enviados como anexo; no entanto, aqui está a lista dos mesmos.

- Login - caso de utilização para criação de conta e subsequente entrada na aplicação.
- AcceptInviteFriendList - caso de utilização que realiza a ação de aceitar um pedido de amizade.
- AcceptInviteChallenge - caso de utilização que realiza a ação de aceitar um convite para um desafio.
- InviteFriendList - caso de utilização que cria um convite entre dois utilizadores.
- InviteChallenge - caso de utilização que cria um convite entre um utilizador e um desafio.
- RefuseInviteFriendList - caso de utilização que realiza a ação de recusar um pedido de amizade.
- RefuseInviteChallenge - caso de utilização que realiza a ação de recusar um pedido de junção a um desafio.

- AddAchievement - caso de utilização que adiciona uma conquista à biblioteca de jogos do utilizador.
- CreateAchievement - caso de utilização que cria uma conquista num jogo da aplicação.
- RemoveAchievement - caso de utilização que remove uma conquista dos jogos da aplicação.
- RemoveAchievementFromChallenge - caso de utilização que que remove uma conquista de um desafio.
- RestartAchievement - caso de utilização que permite ao utilizador recomeçar uma conquista.
- UnlockAchievement - caso de utilização que termina uma conquista na biblioteca de jogos do utilizador.
- UnlockAchievementInChallenge - caso de utilização que termina uma conquista num desafio.
- AddGame - caso de utilização que adiciona um jogo à biblioteca de jogos do utilizador.
- CreateGame - caso de utilização que adiciona um jogo aos jogos da aplicação.
- RemoveGame - caso de utilização que remove um jogo dos jogos da aplicação.
- RemoveGameLibrary - caso de utilização que remove um jogo da biblioteca de jogos da aplicação.
- CreateChallenge - caso de utilização que adiciona um desafio aos desafios da aplicação.
- FinishChallenge - caso de utilização que termina um desafio.
- JoinChallenge - caso de utilização que adiciona a um utilizador a um desafio.

- RemoveChallenge - caso de utilização que remove um desafio dos desafios da aplicação.

- StartChallenge - caso de utilização que começa um desafio.

## 3.2 Fundamentos

Nesta fase serão explicados alguns conceitos considerados fulcrais, usados ao longo do projeto, tais como: Android Studio, MVVM, ViewModel, Parcelable, LiveData; enquadrando os mesmos simultaneamente.

### 3.2.1 Achievement

Termo utilizado na comunidade de jogadores de videojogos que descreve uma proeza ou desafio.

### 3.2.2 Challenge

Termo utilizado para definir um conjunto de Achievements em que um conjunto de jogadores compete desde uma data de começo a uma de finalização.

### 3.2.3 Apollo Server

O Apollo Server é uma biblioteca open-source utilizada para criar APIs GraphQL com JavaScript ou TypeScript. Ele serve como o servidor que processa pedidos GraphQL (queries e mutations), liga-se a fontes de dados (como bases de dados ou APIs externas) e devolve as respostas ao cliente.

### 3.2.4 GraphQL

O GraphQL é uma linguagem de consulta para APIs (Application Programming Interfaces). Serve como alternativa ao modelo REST tradicional. Ao contrário das APIs REST, em que cada endpoint devolve dados fixos, o GraphQL permite que o cliente defina exatamente os dados que quer receber, o que reduz a quantidade de dados transferidos e evita pedidos desnecessários.

### 3.2.5 AddDataSource

Refere-se a uma função/método usada para adicionar fontes de dados externas ao Apollo Server. Estas fontes podem ser APIs REST, bases de dados, etc. No Apollo, esta configuração é feita através da função dataSources, onde se definem classes para aceder e gerir esses dados.

### 3.2.6 TypeORM

O TypeORM é uma biblioteca ORM (Object-Relational Mapping) para JavaScript e TypeScript. Permite interagir com bases de dados relacionais (como PostgreSQL, MySQL, etc.) usando classes e objetos, em vez de escrever SQL diretamente. Cada classe representa uma tabela da base de dados.

### 3.2.7 GraphQLTypes

Os GraphQLTypes (tipos do GraphQL) são os blocos principais de uma estrutura GraphQL. Eles definem os tipos de dados que podem ser utilizados ou devolvidos numa API. Exemplos:

- Tipos escalares: String, Int, Boolean
- Tipos de objetos: User, Post
- Tipos especiais: Query, Mutation, Input, etc.

### 3.2.8 Entity

Uma Entity (entidade) é uma classe que representa uma tabela numa base de dados quando se usa uma biblioteca ORM como o TypeORM. Cada propriedade da classe representa uma coluna. Usa-se para estruturar e mapear os dados de forma organizada.

### 3.2.9 Query

Uma Query em GraphQL é utilizada para ler ou ir buscar dados. Equivale a um pedido GET numa API. O cliente envia uma query a pedir os dados que precisa, e o servidor responde apenas com essa informação.

### 3.2.10 Mutation

Uma Mutation é usada em GraphQL para modificar dados, como criar, atualizar ou apagar registos. Equivale aos métodos POST, PUT ou DELETE nas APIs.

### 3.2.11 Android Studio

O Android Studio é o ambiente de desenvolvimento integrado (IDE) oficial para o desenvolvimento de aplicações Android. Fornece um conjunto completo de ferramentas que facilitam a criação, teste e manutenção de aplicações para dispositivos móveis com o sistema operativo Android.

Entre as suas funcionalidades destacam-se o editor de código inteligente, o emulador de dispositivos Android, ferramentas de análise de desempenho, integração com controlo de versões (como Git), e suporte a múltiplas versões da plataforma Android. O Android Studio tem também um sistema robusto de gestão de dependências através do Gradle.

#### Compose

Nas versões mais recentes, o Android Studio oferece suporte completo ao Jetpack Compose, o novo framework declarativo da Google para a construção de interfaces de utilizador. Este permite definir os componentes visuais diretamente em Kotlin, substituindo o tradicional XML, o que resulta em código mais limpo, conciso e reativo.

### 3.2.12 MVVM

O MVVM (Model-View-ViewModel) é um padrão de arquitetura de software amplamente utilizado no desenvolvimento de aplicações modernas, incluindo aplicações Android. Este padrão tem como principal objetivo separar claramente a lógica da interface gráfica da lógica de negócio e dos dados, promovendo uma estrutura de código mais limpa, testável e de fácil manutenção.

- Model: Representa a camada de dados e lógica de negócio da aplicação. Pode incluir chamadas à API, acesso a bases de dados locais, ou qualquer outra fonte de dados.

- View: Corresponde à interface gráfica apresentada ao utilizador. No caso do Android moderno, esta camada é frequentemente construída com Jetpack Compose, permitindo que o ecrã reaja automaticamente às alterações de estado.

- ViewModel: Atua como intermediário entre a View e o Model. Contém a lógica de apresentação e expõe os dados (normalmente através de State, LiveData ou Flow) que a View consome. O ViewModel é responsável por preparar os dados para apresentação, mantendo a View o mais simples possível.

**LiveData**

LiveData é uma classe observável fornecida pela biblioteca Android Architecture Components. Foi criada para armazenar e observar dados de forma reativa, respeitando o ciclo de vida dos componentes Android, como Activities.

O LiveData permite que a interface gráfica (UI) observe mudanças nos dados sem necessidade de fazer atualizações manuais. Quando o valor de um LiveData é alterado, todos os observadores ativos (por exemplo, uma Activity que está visível) são notificados automaticamente.

### 3.2.13 Activity

Uma Activity é um dos componentes principais de uma aplicação Android. Representa um único ecrã (interface gráfica) com o qual o utilizador pode interagir — semelhante a uma "janela"de uma aplicação de computador. Cada vez que um utilizador abre uma nova parte da app (por exemplo, um ecrã de login ou de perfil), uma nova Activity pode ser iniciada.

No ecossistema moderno do Android, com Jetpack Compose, o MVVM integra-se naturalmente com o paradigma declarativo. O ViewModel gere o estado da UI e comunica alterações de forma reativa, permitindo que a interface se actualize automaticamente sempre que os dados mudem. Esta abordagem reduz significativamente a complexidade do código e melhora a escalabilidade da aplicação.

## 3.3 Abordagem

### 3.3.1 Características da APP

Como foi explicado anteriormente a nossa aplicação serve para utilizadores poderem criar jogos e conquistas desses jogos para se entreterem a terminar essas conquistas, tanto individualmente (adicionando essas conquistas à biblioteca de jogos pessoal), como competindo com outros utilizadores. A aplicação permite:

- Criar Jogos, Conquistas e Desafios.
- Eliminar Jogos, Conquistas e Desafios.
- Editar Jogos, Conquistas, Desafios e o Perfil do utilizador.
- Adicionar Jogos criados (vêm com todas as conquistas criadas desse jogo) à biblioteca de jogos pessoal do utilizador.
- Adicionar Conquistas aos Desafios.
- Terminar Conquistas, tanto na biblioteca de jogos pessoal como nos desafios.
- Convidar Utilizadores para se juntarem à sua lista de amigos.
- Convidar Utilizadores para se juntarem a um Desafio ou vice-versa; adicionando-os.
- Aceitar ou Recusar o convite de outros utilizadores.
- Começar um Desafio
- Competir até as condições do Desafio forem completas e de seguida terminar o Desafio.

Os jogos, as conquistas e os desafios só podem ser editados pelo utilizador que os criou. A edição do perfil só pode ser feita na página de utilizador com o mesmo Id de quem deu LogIn. Começar o desafio só é possível por quem o criou. As condições de terminar um desafio podem ser por exemplo "o primeiro a terminar todas as conquistas". Quando isso acontecer os pontos são atribuídos com base na quantidade de conquistas terminadas (1 to 1 entre conquistas e pontos).

### 3.3.2 Arquitetura do Projeto

A figura seguinte mostra a arquitetura utilizado no projeto, pode-se ver as tecnologias utilizadas ao longo do desenvolvimento do mesmo e, desta forma, analisar como foi feito o processo desde a fase inicial de modelação e desenho de interface até à fase final.

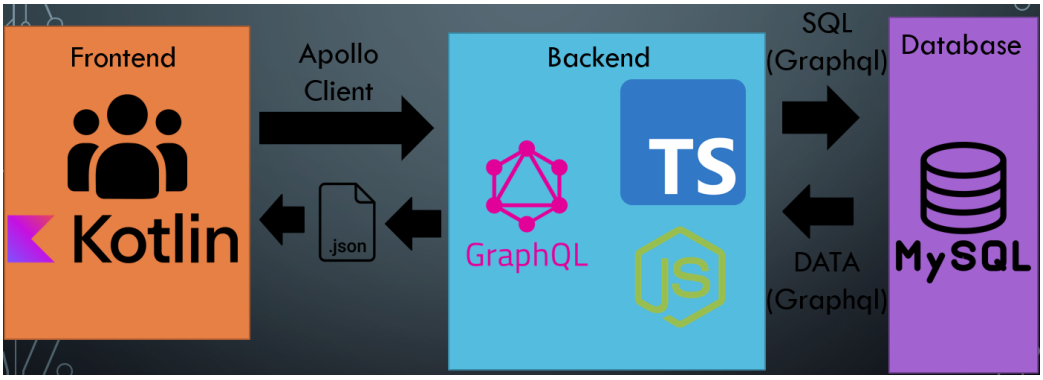


Figura 3.5: Aquitetura do Projeto

**Frontend (à esquerda)**

- Linguagem de Programação: Kotlin
- Plataforma: Android (Android Studio)
- Cliente: Usa Apollo Client para comunicar com o backend.
- Formato de dados: Envia e recebe dados em JSON (Utilizando ficheiros .graphql com a indicação de como fazer pedidos e receber pedidos a API a partir de um ficheiro schema.graphql).

A aplicação móvel desenvolvida em Kotlin faz pedidos ao backend através do Apollo Client, que é uma biblioteca para consumir APIs GraphQL.

**Backend (ao centro)**

Tecnologias Usadas:

- GraphQL (para definir a API e intermediar as comunicações)

- TypeScript (TS)
- Node.js (representado pelo logo JS em hexágono)

O backend está implementado em TypeScript e corre com Node.js. Ele usa GraphQL para processar:

- Queries (para buscar dados)
- Mutations (para modificar dados)

Este backend age como intermediário entre o frontend e a base de dados.

**Database (à direita)**

- Base de dados: MySQL
- O backend comunica com o MySQL, usando a biblioteca TypeORM para fazer consultas e alterações na base de dados.

O backend traduz os pedidos GraphQL em SQL para interagir com a base de dados. Os dados são lidos e devolvidos em formato GraphQL, que o Apollo Client interpreta.

**Em Conclusão**

O utilizador interage com a app (Kotlin).

A app usa Apollo Client para enviar um pedido GraphQL (em JSON) ao backend.

O backend (Node.js + TypeScript) processa o pedido, consulta a base de dados (MySQL) se necessário.

Os dados são transformados em resposta GraphQL e enviados de volta ao frontend.

## 3.4 Planeamento

Como se pode ver pela imagem acima:

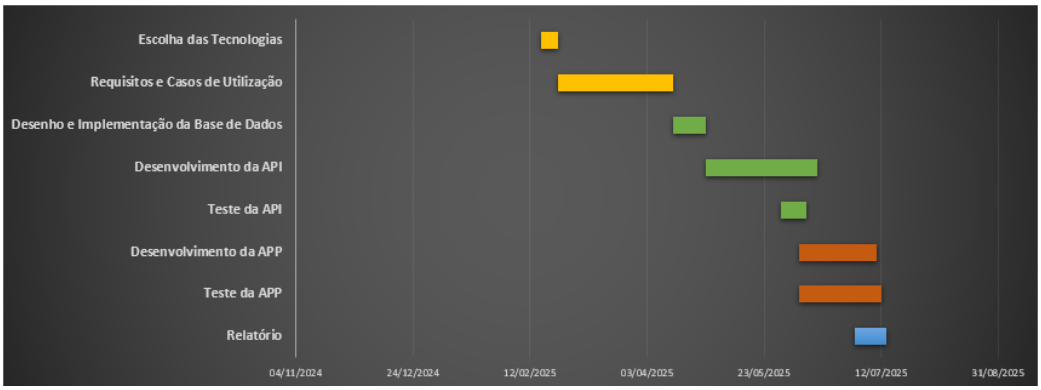


Figura 3.6: Planeamento

- Para este trabalho começámos por contextualizar o desenvolvimento deste projeto, que demorou uma semana, sendo o projeto escolhido por nós.
- Com a ideia formularizada realizámos os requisitos e os casos de utilização, que depois de alguma discussão terminámos nas Férias da Páscoa.
- De seguida criámos o modelo de entidade e associaçãoA.1 e o planeamento dos métodos. Apesar de ter sido mais ou menos discutido até ao final, tirando um acréscimo aqui e ali demorou cerca de duas semanas.
- Construímos a aplicação e a API mais ou menos no mesmo tempo, no entanto a App começou a ser desenvolvida ligeiramente mais tarde e, como a API era tecnologia não explorada previamente, acabou por demorar um pouco mais.
- Terminando nos testes e resolução de problemas, bem como o Relatório que os dois juntos demoraram cerca de duas semanas.

# Capítulo 4

# Implementação do Modelo

Neste capítulo será abordado a implementação do modelo proposto no capitulo anterior, ao longo do capitulo sera explicado com detalhe as decisões tomadas e o processo utilizado para desenvolver cada componente.

### 4.0.1 Dependências Tecnológicas

No desenvolvimento do projeto foram utilizados diversos softwares de modo a conseguir alcançar os objectivos propostos. Como dito anteriormente as maiores tecnologias usadas foram o Kotlin, a tecnologia onde a aplicação foi construida, o NodeJS, a tecnologia onde a API foi realizada e o MySQL, a tecnologia onde a Base de Dados persiste, contudo foi necessário utilizar vários software para obter mais algum controlo de funcionalidade, para tal foi usado para a API o Typescript e o GraphQL.

#### Dependências tecnológicas utilizadas pela APP:

#### Kotlin

Para o desenvolvimento da aplicação móvel, foi utilizada a linguagem **Kotlin**. Kotlin é uma linguagem de programação moderna, concisa e segura, desenvolvida pela JetBrains e oficialmente suportada pelo Android desde 2017. Foi concebida para interoperar totalmente com Java, mas oferece uma sintaxe mais clara e funcionalidades mais avançadas. No nosso projeto, Kotlin foi utilizado para construir toda a lógica da aplicação Android, incluindo a interface gráfica com **Jetpack Compose**, a comunicação com a API, e a

gestão de dados locais e remotos. A escolha por Kotlin permitiu um desenvolvimento mais rápido, com menos erros e uma base de código mais limpa e sustentável.

### Compose

O Plugin **Compose** habilita o uso do **Jetpack Compose** com o compilador **Kotlin**, fornecendo o suporte necessário para construção de interfaces gráficas reativas e declarativas. Este plugin integra ferramentas de compilação específicas que permitem utilizar anotações como **@Composable**, além de permitir pré-visualizações da UI no **Android Studio**. Ele é indispensável em projetos que adotam o **Jetpack Compose** como base para construção da interface.

### Lifecycle ViewModel e LiveData KTX

As bibliotecas **lifecycle-viewmodel-compose** e **lifecycle-livedata-ktx** fornecem **integrações específicas** do ciclo de vida da aplicação com **Kotlin**. Elas permitem a utilização de padrões arquiteturais como **MVVM (Model-View-ViewModel)** de forma idiomática, segura e eficiente. A versão -ktx oferece extensões Kotlin que reduzem a verbosidade e aumentam a legibilidade do código. Em conjunto, essas dependências facilitam a separação entre lógica de negócio e interface de utilizador, além de permitir uma gestão eficiente de estados e dados observáveis.

### androidx.core:core-ktx

Esta biblioteca fornece um **conjunto de extensões Kotlin** para as classes principais da plataforma **Android**, como **SharedPreferences**, **Context**, **Bundle**, entre outras. Seu objetivo é tornar o uso dessas **APIs** mais idiomático e alinhado aos princípios da linguagem **Kotlin**. Ela substitui chamadas verbosas por sintaxes mais concisas, promovendo um código mais legível e menos propenso a erros.

### Firebase Auth com suporte a Kotlin (firebase-auth-ktx)

A biblioteca **firebase-auth-ktx** fornece **extensões Kotlin** para a **integração com o Firebase Authentication**, simplificando tarefas como lo-

gin, registro de usuários e gerenciamento de sessão. O uso da versão -ktx permite **aproveitar os recursos da linguagem Kotlin**, como corrotinas e lambdas, tornando o código mais conciso, seguro e fácil de manter.

### Plugin kotlin-parcelize

O plugin **kotlin-parcelize** permite utilizar a anotação **@Parcelize**, que automatiza a implementação da interface **Parcelable**. Isso é especialmente útil ao **transferir objetos entre componentes Android**, como **Activities** ou **Fragments**, sem a necessidade de escrever manualmente o boilerplate (código repetitivo, padronizado e geralmente obrigatório que precisa de ser escritos mesmo que não adicione lógica nova ou específica ao programa.) típico de **Parcelable**. Esse plugin contribui para a redução de código repetitivo e aumento da produtividade.

### NodeJS

O NodeJS foi a technologia que se escolheu para desenvolver a API, esta technologia foi criada por Ryan Dahl em 2009 com o objectivo de desenvolver a funcionalidade do JavaScript podendo ser utilizada do lado do Servidor. O NodeJS foi inicialmente escolhido devido a ser uma tecnologia que nunca estivemos em contacto com e a nossa vontade de tentar criar algo do 0. O NodeJS é uma tecnologia que oferece alta escalabilidade e desempenho devido à sua arquitetura orientada a eventos e I/O não bloqueante. O vasto ecossistema de pacotes e bibliotecas, juntamente com uma comunidade ativa, também contribuem para um desenvolvimento mais rápido e eficiente de aplicações utilizando esta tecnologia.

### TypeScript

Para ajudar na definição de tipos e controlar melhor o tipo de informação utilizada na API decidimos utilizar TypeScript. O TypeScript é uma linguagem de programação desenvolvida pela Microsoft, que se baseia em JavaScript e adiciona tipagem estática opcional ao código. Essa tipagem ajuda a detectar erros em tempo de desenvolvimento e oferece uma experiência de desenvolvimento mais robusta. Na nossa API utilizamos TypeScript para por exemplo defenir tipos de retorno específicos para cada atributo de cada entidade ou

até para criar os nossos próprios tipos como "**Message**"e "**CustomDate**".

### 4.0.2 GraphQL

Para gerir a criação, e transmissão de informação na API utilizamos na tecnologia GraphQL. GraphQL é uma linguagem de consulta para APIs e um ambiente de execução para essas consultas, desenvolvido originalmente pelo Facebook em 2012 e lançado como open-source em 2015. O seu principal objetivo é oferecer uma forma mais eficiente, flexível e intuitiva de interagir com dados em aplicações web e móveis. Ao contrário do modelo tradicional REST, em que cada endpoint retorna dados fixos, o GraphQL permite ao cliente especificar exatamente os dados que pretende receber. Isto reduz o excesso de dados transferidos (overfetching) ou a falta de informação (underfetching), comuns em APIs REST. No nosso projeto, o GraphQL foi utilizado para realizar desde a criação da BD até toda a lógica de transmissão de dados "queries"existentes na API.

**Dependências tecnológicas utilizadas pela API:**

- **Apollo Server**: Biblioteca para criar e gerir APIs GraphQL no lado do servidor (Node.js).
- **Cors**: Biblioteca que permite configurar o acesso entre domínios diferentes, essencial para permitir que o frontend comunique com o backend de forma segura.
- **DotEnv**: Biblioteca que carrega variáveis de ambiente a partir de um ficheiro `.env`, permitindo configurar dados sensíveis (por exemplo as credencias de acesso ao Apollo Server) fora do código-fonte.
- **Multer**: Biblioteca usada para processar uploads de ficheiros em aplicações Node.js, utilizado para upload de imagens na API.
- **Nodemon**: Ferramenta de desenvolvimento que reinicia automaticamente a aplicação Node.js sempre que há alterações no código-fonte. Facilita o processo de testes e desenvolvimento.

- **Ts-Node**: Ferramenta que permite executar ficheiros TypeScript diretamente no ambiente Node.js, sem necessidade de compilação manual para JavaScript.

- **TypeORM**: Biblioteca ORM (Object-Relational Mapping) para TypeScript e JavaScript que facilita a interação com bases de dados relacionais, permitindo trabalhar com entidades e queries de forma orientada a objetos.

- **Express**: Framework leve para Node.js usada para criar servidores web e definir rotas da aplicação.

- **Rover**: Ferramenta utilizada para extrair o ficheiro schema.graphql que descreve todas as regras para que as Entidades, queries e mutation sejam recriadas no nosso caso na nossa APP para possível ligação ao kotlin.

### MySQL

Para gerir a criação e manipulação dos dados na nossa aplicação, utilizamos a base de dados relacional **MySQL**. O MySQL é um sistema de gestão de bases de dados (SGBD) amplamente utilizado, conhecido pela sua robustez, performance e escalabilidade. É open-source e baseado em SQL (Structured Query Language), a linguagem padrão para gerir e consultar dados relacionais. No nosso projeto, o MySQL foi utilizado para armazenar de forma estruturada todas as entidades e relacionamentos da aplicação, garantindo a integridade e consistência dos dados. A interação com a base de dados é feita através de um ORM (TypeORM), que facilita a criação, leitura, atualização e eliminação dos dados (CRUD), traduzindo comandos em TypeScript para queries SQL otimizadas para o MySQL.

**Docker** Para garantir a portabilidade e facilidade de implantação da aplicação, utilizamos o **Docker**. Docker é uma plataforma que permite criar, distribuir e executar aplicações dentro de containers, que são ambientes isolados e leves que garantem que o software funcione da mesma forma em qualquer sistema. Na nossa arquitetura, a **API** e a base de dados **MySQL** correm em containers distintos, garantindo isolamento e independência entre os serviços.

Esta separação permite uma maior flexibilidade na gestão dos recursos e facilita a escalabilidade da aplicação. A comunicação entre os dois containers é feita por uma rede interna gerida pelo Docker, garantindo desempenho e segurança.

### 4.0.3 Implementação da APP

Para a **implementação da APP**, o desenvolvimento da mesma foi realizada inteiramente utilizando o **Android Studio com a programação em Kotlin**. Tendo este programa total suporte à criação da aplicação não foi necessário a utilização de outro programa de suporte.

A visualização dos ecrãs é organizada usando Compose que é uma biblioteca do Kotlin que permite a criação de campos de texto, listas, botões etc..., servindo como uma substituição aos layouts de xml utilizados anteriormente no Android Studio, tornando a criação de páginas mais dinâmica.

#### Conceitos e Explicações Necessárias

Nas imagens das páginas que se seguem serão visualizados vários tipos de componentes de **Screens** ou páginas como foram referidas. Segue aqui uma explicação dos mais utilizados.

- Text: Componente utilizado para exibir **texto na interface gráfica**.
- Button: Elemento interativo que **reage ao clique do utilizador**. É usado para executar ações ou navegar entre páginas. Pode conter texto, ícones ou ambos. Sempre que é referido abaixo **Button** é um botão destes.
- Image: Permite **mostrar imagens estáticas na UI**, vindas de recursos (painterResource), URLs (com bibliotecas externas como Coil), ou Bitmap. Todas as imagens que se encontram nas páginas abaixo.
- Card: Um **componente estilizado com sombra e cantos arredondados**, usado para agrupar conteúdo de forma visualmente destacada. Suporta personalização de cor, elevação, forma e padding. Elementos das nossas listas.

- OutLineTextField: Campo de **input de texto com borda visível** (estilo Material Design). Suporta funcionalidades como placeholder, validação, ícones internos, e eventos como onValueChange (permite alterar o texto em tempo real). Usado para as **SearchBars** mencionadas abaixo.

- DropDownMenu: Componente que exibe uma lista suspensa de opções (menu) ancorada a outro componente (normalmente um botão ou ícone). Ideal para mostrar ações ou escolhas num espaço compacto. É controlado por um estado booleano (expanded) e requer uma função onDismissRequest. Usado na **TopBar** e na procura dos jogos e achievements.

- LazyColumn: Componentes de **lista preguiçosa** que carregam apenas os itens visíveis no ecrã, otimizando a performance. Usado para listas grandes e dinâmicas. Sempre que for mencionada lista abaixo refe-se a este componente.

- AlertDialog: Janela utilizada para mostrar mensagens importantes ou confirmar ações do utilizador. Suporta título, texto, botões de confirmação e cancelamento, e ações associadas. Serve para confirmações do LogOut, de eliminação de conteúdo e de gerência de emails.

### Authentication

Tirando a aplicação realizamos o **SignIn** e o **SignUp** utilizando a **Autenticação (Authentication) do Firebase** para guardar os emails e passwords dos utilizadores. Abaixo pode-se ver como se realiza a instância e como se realiza a busca de informação tanto para criar conta como para realizar entrar na aplicação utilizado essa conta.

```
private val firebaseAuth = FirebaseAuth.getInstance()
```

Figura 4.1: Instância do Firebase

```
firebaseAuth.createUserWithEmailAndPassword(username, password)
```

Figura 4.2: SignUp usando Authentication

```
firebaseAuth.signInWithEmailAndPassword(username, password)
```

Figura 4.3: SignIn usando Authentication

### Utilização de Object MyId

Na Realização da APP, precisou-se no início da utilização de uma variável de controlo que foi a classe MyId. Esta classe manteu-se presente até ao final e é responsável por guardar a informação do utilizador em todas as páginas da aplicação e é afetada quando o SignIn é feito. Em Kotlin, o tipo object serve para definir um singleton, ou seja, um objeto único que é criado automaticamente e só existe uma vez durante a execução do programa.

É muito útil para:

- Criar utilitários (funções reutilizáveis)
- Armazenar constantes globais
- Gerenciar recursos compartilhados

```
object MyId{
    var user: User? = null
```

Figura 4.4: Singleton Object MyId e variável user

```
val userProfile = MyId.user
```

Figura 4.5: Utilização do MyId numa classe

### Extrutura Modular de Integração com AppDomain

Para testar a aplicação com a utilização de lógica modular, foi realizada a ligação com a **AppDomain** que foi um ficheiro criado localmente para tentar conter a mesma lógica de fluxo que a API (mencionada mais à frente). Apesar de ter funcionado, não teve tanta utilização, visto que só quando as transições entre páginas são feitas é que foi corrido o **AppDomain**. Devido a falta de bidirecionalidade de entidades (entidades com listas de entidades

dentro), não era fácil o acesso a determinadas páginas por isso rapidamente foi esquecido pela API.

Exemplo de bidirecionalidade:

- Um User tem muitos Challenges (challenges)
- Um Challenge tem muitos Users (participants)
- Ou seja, a relação é bidirecional.

### Relação entre ViewModel, Screen e Activity

Nesta ligação a Activity é responsável pela execução dos métodos, o ViewModel vai buscar os métodos à API (será referida mais à frente) e o Screen representa a UI da aplicação.

A Activity mostra o Screen e quando um utilizador interage com o UI (carrega num botão, escreve num search...) a Activity executa as funcionalidades necessárias dadas pelo ViewModel(métodos).

Exemplo:


Figura 4.6: Utilização do MyId numa classe

### Organização das Páginas da Aplicação

Antes da explicação das páginas, explicarse-á a criação da **TopBar** e da **BottomBar**. Estes **Screens** realizados utilizando **Compose**, são utilizados em todas as páginas para facilitar a navegação da aplicação.

### TopBar

A TopBar é composta por dois **Ícones** que realizam a navegação para a página de **About** (Ícone da Esquerda) e para a página de **Perfil** (Ícone da Direita). Permite também no Ícon da Esquerda realizar o **Logout** que aparece como um **Screen de PopUp** que pede para confirmar a saída da App, navegando para a página de **SignIn**.

### BottomBar

A BottomBar é composta por cinco **Ícones** que realizam a navegação para as páginas **Library**, **Games**, **Users**, **FriendList**, **Challenges** (Ícones organizados da Esquerda para a Direita).

Para a realização da APP dividimo-la em 5 secções:

- **Library**: Acesso aos jogos da biblioteca pessoal e onde se pode aceder às proezas dos mesmos e terminá-las/recomeçar-las. Pode-se também remover qualquer jogo, removendo as proezas relacionadas a esse jogo na biblioteca pessoal.

- **Games**: Acesso aos jogos e proezas da aplicação e onde os mesmos podem ser criados, editados, eliminados e adicionados à biblioteca pessoal.

- **Users**: Acesso à lista total de outros utilizadores da aplicação, bem como o seu perfil. É aqui que se pode também convidar qualquer utilizador para a nossa lista de amigos ou para um desafio que se tenha criado.

- **FriendList**: Acesso à lista de utilizadores que aceitaram os pedidos de amizade, bem como o seu perfil. Pode-se também gerir os pedidos de amizade enviados.

- **Challenges**: Acesso aos desafios onde o utilizador foi adicionado. Pode-se também ver outros utilizadores e o seu progresso, bem como as proezas adicionadas ao desafio. Quem for o criador do desafio pode também adicionar e remover utilizadores e proezas.

De seguida serão explicadas as secções acima e pela ordem que faz sentido com o que é representado em cada página:

## Resumo das Páginas da Aplicação (Jetpack Compose)

### Library

- **Biblioteca Pessoal**: Lista os jogos do utilizador, com pesquisa por nome na `SearchBar`. Ao clicar num jogo, navega-se para a página `Game`.
- **Game da Library**: Exibe detalhes do jogo e suas proezas. Permite remover o jogo da biblioteca (`Button Remove Game`) e aceder a uma proeza (`Achievement`).
- **Achievement da Library**: Mostra os detalhes da proeza, permitindo terminá-la (`Button Finish Achievement`) ou reiniciá-la (`Button Restart Achievement`).

### Games

- **Games**: Lista todos os jogos com filtros por nome, consola, editor, desenvolvedor e género (`SearchBar`). Permite criar novos jogos e aceder ao `Game`.
- **Game dos Games**: Semelhante ao `Game da Library`, mas com botões diferentes: adicionar à biblioteca (`Button Add Game`), editar (`Button Edit Game`), eliminar (`Button Eliminate Game`) e criar proezas (`Button Create Achievement`), todos acedidos apenas pelo criador com a excessão do (`Button Add Game`).
- **Achievement**: Igual ao `Achievement da Library`, mas com funções de edição (`Button Edit Achievement`) e eliminação (`Button Eliminate Achievement`), caso o utilizador seja o criador.

### Utilizadores

- **Users / UsersByScore / FriendList**: Todas estas páginas listam utilizadores, com filtros (`SearchBar`). A `UsersByScore` permite comparar `Total Points`. A `FriendList` mostra apenas amigos.

- **User** (inclui `User dos Users`, `User da FriendList`, `User de Challenge`): Página de perfil do utilizador, com os desafios em que participa. Dependendo do contexto, permite convidar (`Button Invite User`) ou remover (`Button Kick User`).

### Convites

- **FriendInvites**: Mostra convites de amizade em estado `Pending` ou `Rejected`. Os convites aceites tornam-se automaticamente amigos.
- **ChallengesInvites**: Idêntica à anterior, mas aplicada a desafios. Mostra convites enviados ou recebidos, e o seu estado.

### Challenges

- **Challenges**: Lista todos os desafios disponíveis. Permite criar novos desafios (`Button CreateChallenge`) ou aceder ao `Challenge`.
- **Challenge** (inclui `Challenge de Challenges`, `Challenge de User`): Exibe detalhes do desafio e participantes, com pesquisa. O criador pode editar (`Button Edit Challenge`), sair (`Button Leave Challenge`), eliminar (`Button Eliminate Challenge`) ou adicionar utilizadores (`Button Add User`). Também permite pedir entrada (`Button Join Challenge`).
- **ChallengeAchievements**: Lista as proezas associadas ao desafio, com filtro por nome. O criador pode adicionar novas (`Button Add Achievement`).

### Criação e Edição

- **CreateEditPages**: Páginas com `SearchBars` para introdução ou edição de dados. Ao finalizar (`Button Finish`), cria-se ou atualiza-se a respetiva entidade (jogo, proeza, desafio, etc.).

**Outras Páginas**

- **SignIn**: Páginas com campos de texto para introdução de `Email` e `Password`. Ao clicar no botão de login (`Button SignIn`), vai-se buscar a conta com o respetivo `Email` e `Password` e entra-se na aplicação. Se não tiver conta pode clicar no texto abaixo e segue para a criação de conta `SignUp`.

- **SignUp**: Páginas com campos de texto para introdução de `Email`, `UserName`, `Password` e `ConfirmPassword`. Ao clicar no botão de signup (`Button SignUp`), cria-se uma conta com o respetivo `Email`, `Password` e `UserName`. Se já tiver conta pode clicar no texto abaixo e segue para o log-in `SignIp`.

- **About**: Página com informação sobre a App.

### 4.0.4 Implementação da API

Um dos primeiros pontos abordados no desenvolvimento da aplicação foi a forma como iríamos estruturar a comunicação entre o frontend e a base de dados. Depois de analisadas várias possibilidades, foi decidido implementar uma **API com recurso à tecnologia GraphQL**, uma vez que esta permite uma maior flexibilidade no envio e receção de dados, especialmente útil para aplicações com estruturas complexas e dinâmicas.

Após essa decisão, o passo seguinte foi definir as tecnologias e ferramentas que iríamos utilizar na implementação da API. Optou-se por **Node.js** como ambiente de execução, **Apollo Server** como motor principal da API GraphQL, e **TypeORM** para a ligação e manipulação da base de dados relacional **MySQL**.

Com esta base tecnológica definida, avançámos para a construção do esquema da API (schema), das queries e mutations, bem como das ligações com a base de dados. Nos pontos seguintes será descrito em mais detalhe o processo de implementação da API e a forma como a comunicação entre os vários componentes da aplicação foi organizada.

### Inicialização da Aplicação e Configuração do Servidor

Na fase inicial da aplicação, é feita a configuração da ligação à base de dados e a preparação do servidor web que irá expor a API GraphQL, bem como gerir o upload e o acesso a imagens.

O processo começa com a configuração do **DataSource** utilizando a biblioteca **TypeORM**, onde são definidos os parâmetros de ligação à base de dados MySQL (como `host`, `username`, `password`, `database`, etc.). As entidades (como `Users`, `Games`, `Achievements`, entre outras) são registadas para que o ORM consiga mapear corretamente as tabelas da base de dados.

```
await appDataSource.initialize();
```

Figura 4.7: Inicialização da appDateSource

Após a base de dados estar ligada com sucesso, é criado um servidor **Express**, que será responsável por:

- Ativar o **CORS**, permitindo o acesso entre domínios diferentes;
- Usar `express.json()` para interpretar automaticamente pedidos com corpo em JSON;
- Servir ficheiros estáticos da pasta `Images` através do endpoint `/Images`;
- Gerir uploads de ficheiros através de um endpoint POST para `/Images`.

Em seguida, é instanciado o **Apollo Server** com o schema GraphQL já definido. A integração com o Express é feita através do middleware `expressMiddleware`, ficando a API GraphQL acessível através do endpoint `/graphql`.

Por fim, o servidor é iniciado na porta definida (por padrão, 3000), ficando pronto para receber pedidos HTTP tanto da API como de ficheiros.

### Definição do Schema GraphQL

Uma parte central da implementação da API foi a construção do **schema GraphQL**, onde são definidas todas as operações disponíveis para o cliente, tanto para leitura (queries) como para escrita ou modificação de dados (mutations).

O schema é composto por dois blocos principais:

**1. RootQuery** A `RootQuery` agrega todas as operações de leitura disponíveis na API. Cada campo corresponde a uma query específica, agrupada por tipo de entidade (Users, Games, Achievements, Challenges, Friends, etc.).

Exemplos de operações incluídas:

- `getAllUsers`: devolve todos os utilizadores;
- `getAllUserFriends`: devolve a lista de amigos de um utilizador.

Estas queries permitem que o cliente aceda aos dados existentes na base de dados de forma controlada e estruturada.

**2. Mutation** A `Mutation` define todas as operações de escrita ou alteração de dados, como criar, atualizar ou eliminar entidades. Estão igualmente organizadas por entidade, incluindo por exemplo:

- `createUser`, `deleteUser`, `updateUser`: para gerir utilizadores;
- `inviteFriend`, `acceptFriendInvite`: para gerir pedidos de amizade;

Cada uma destas operações é mapeada para uma função (resolver) responsável por executar a lógica associada.

**3. Exportação do Schema** Por fim, todo o schema é reunido com:

```
export const schema = new GraphQLSchema({
    query: RootQuery,
    mutation: Mutation
});
```

Figura 4.8: Criação do Schema GraphQL

Isto permite ao servidor Apollo disponibilizar a API completa, tornando estas operações acessíveis ao cliente através de um único endpoint GraphQL (`/graphql`).

### Definição das Entidades

Para representar e organizar os dados da aplicação de forma estruturada, foram definidas diversas **entidades** com recurso à biblioteca **TypeORM**. Cada entidade corresponde diretamente a uma tabela na base de dados relacional (MySQL), e é representada no código através de classes decoradas com `@Entity()`.

Cada uma destas classes contém um conjunto de campos com anotações que descrevem as suas propriedades e restrições — por exemplo:

- `@PrimaryGeneratedColumn()` define uma chave primária gerada automaticamente;
- `@Column({unique: true})` especifica que o valor de um campo tem de ser único;
- `@Column({nullable: true})` permite que o campo seja opcional na base de dados.

Além dos atributos simples (como nomes, datas, ou pontuações), as entidades incluem também as **relações entre si**, como:

- `@OneToMany` e `@ManyToOne` para relações de um-para-muitos;
- `@ManyToMany` (quando aplicável), com tabelas intermédias.

Estas relações permitem representar ligações lógicas entre utilizadores, jogos, conquistas, desafios, amigos, entre outros, de forma que o ORM consiga gerar e sincronizar corretamente a estrutura da base de dados.

O uso do **TypeORM** facilita a criação, leitura e atualização destes dados com código orientado a objetos, abstraindo a escrita direta de SQL e mantendo a coerência entre o modelo de dados e a lógica da aplicação.

### Definição dos Tipos de Dados (GraphQL Object Types)

Após definidas as entidades da base de dados, o passo seguinte na construção da API foi criar os **tipos de dados** utilizados no schema GraphQL. Para isso, recorreu-se ao uso de **GraphQLObjectType**, que define a estrutura dos objetos retornados e aceites pela API.

Cada tipo representa uma entidade da aplicação (como `User`, `Game`, `Challenge`, etc.) e define os campos que podem ser consultados ou manipulados..

Estes campos são tipados com os tipos nativos do GraphQL, como:

- `GraphQLString` – para textos;
- `GraphQLInt` – para números inteiros;
- `GraphQLID` – para identificadores únicos;
- `GraphQLList` – para listas de objetos relacionados.

Além disso, para campos de data como `MemberSince` e `LastLogin`, foi definido um tipo personalizado (`CustomDate`) para garantir um formato consistente e correto.

Alguns campos que representam **relações com outras entidades** (como `UserAchievements`, `UserGames`, etc.) são definidos como listas (`GraphQLList`) de outros tipos de objeto. Nestes casos, foram também incluídas **funções de `resolve| personalizadas`**, que tratam a forma como os dados são carregados, garantindo que mesmo campos nulos ou não carregados devolvem listas vazias em vez de `undefined`.

Este processo de mapeamento entre entidades TypeORM e tipos GraphQL permite estruturar claramente os dados da API, garantindo validação, consistência e clareza tanto para o lado do servidor como para os clientes que consomem a API.

Para alem dos tipos nativos de GraphQL foram tambem criados 2 tipos personalizados:

- **CustomDate**: Tipo escalar personalizado utilizado para representar datas em formato ISO (`YYYY-MM-DDTHH:mm:ss.sssZ`) na API GraphQL. Permite converter automaticamente strings recebidas em objetos `Date` no backend, e garantir que as datas são sempre devolvidas com o mesmo formato padronizado.
- **Message**: Tipo de objeto utilizado como resposta padrão em mutations. Contém dois campos: `successfull` (booleano) para indicar se a operação foi bem-sucedida, e `message` (string) para fornecer uma descrição ou feedback sobre o resultado da operação.

**Lista e defenição das diferentes queries presentes na API:**

Tabela 4.1: Resumo das Queries da API GraphQL

| **Query** | **Descrição** |
|---|---|
| getAllUsers | Devolve todos os utilizadores existentes na base de dados. |
| getUserByUserId | Obtém os dados de um utilizador específico através do seu ID. |
| getUserUserGames | Obtém todos os jogos relacionados a um utilizador. |
| getUserUserChallengeAchievements | Obtém todas as proezas de desafios associadas a um utilizador. |
| getUserUserAchievements | Obtém todas as conquistas associadas a um utilizador. |
| getAllUsersByTotalPoints | Retorna utilizadores ordenados pelos seus pontos totais. |
| getAllUserFriends | Lista os amigos de um determinado utilizador. |
| getAllUsersInChallenge | Retorna utilizadores que participam num desafio específico. |
| getAllUsersNotInChallenge | Retorna utilizadores que não participam num desafio específico. |
| searchUsers | Pesquisa utilizadores com base em critérios fornecidos. |
| searchUserGames | Pesquisa jogos associados a utilizadores. |
| getAllGames | Devolve todos os jogos disponíveis. |
| getGameByGameId | Obtém detalhes de um jogo específico. |
| searchGames | Pesquisa jogos pelo nome ou outros critérios. |
| getAllAchievements | Lista todas as conquistas existentes. |

*Continua na página seguinte*

| **Query** | **Descrição** |
|---|---|
| getAchievementByAchievementId | Obtém detalhes de uma conquista específica. |
| getAllAchievementsNotInAChallenge | Retorna conquistas não associadas a desafios. |
| searchAchievements | Pesquisa conquistas por nome ou descrição. |
| getUserGameByUserGameId | Obtém um jogo relacionado a um utilizador. |
| getUserAchievement ByUserAchievementId | Obtém uma conquista relacionada a um utilizador. |
| getAllUserChallengeInvites | Lista todos os convites de desafio relacionados a um utilizador (enviados ou recebidos). |
| getAllChallengeInvitesByChallengeId | Lista todos os convites de um desafio específico. |
| getAllJoinableChallenges | Retorna desafios disponíveis para entrada. |
| getChallengeByChallengeId | Obtém detalhes de um desafio específico. |
| searchChallenges | Pesquisa desafios pelo nome ou outros critérios. |
| getUserChallengeByUserChallengeId | Obtém um desafio relacionado a um utilizador. |
| getUserChallengeAchievement ByUserChallengeAchievementId | Obtém uma conquista de um utilizador relacionada a um desafio. |
| getChallengeAchievement ByChallengeAchievementId | Obtém uma conquista associada a um desafio. |
| getAllFriendInvitesByReceiverId | Lista convites de amizade recebidos por um utilizador. |
| getAllFriendInvitesBySenderId | Lista convites de amizade enviados por um utilizador. |

*Continua na página seguinte*

| **Query** | **Descrição** |
|---|---|
| getAllFriendInvites | Lista todos os convites de amizade. |
| searchFriends | Pesquisa amigos com base em critérios fornecidos. |

**Lista e definição das diferentes mutations presentes na API:**

Tabela 4.2: Resumo das Mutations da API

| **Mutation** | **Descrição** |
|---|---|
| createUser | Cria um novo utilizador com os dados fornecidos. |
| deleteUser | Remove um utilizador da base de dados. |
| updateUser | Atualiza os dados de um utilizador existente. |
| logIn | Autentica um utilizador e inicia sessão. |
| createGame | Adiciona um novo jogo ao sistema. |
| deleteGame | Remove um jogo existente. |
| updateGame | Atualiza os dados de um jogo. |
| addGame | Associa um jogo a um utilizador. |
| removeGame | Remove a associação de um jogo a um utilizador. |
| createAchievemet | Cria uma nova conquista. |
| deleteAchievemet | Elimina uma conquista. |
| updateAchievement | Atualiza os dados de uma conquista. |
| lockUnlockAchievement | Bloqueia ou desbloqueia uma conquista. |
| createChallengeAchievement | Associa uma conquista a um desafio. |
| deleteChallengeAchievement | Remove uma conquista de um desafio. |
| lockUnlockChallengeAchievement | Bloqueia ou desbloqueia uma conquista num desafio. |
| inviteUserToChallenge | Envia convite para um utilizador participar num desafio. |
| acceptChallengeInvite | Aceita um convite de desafio. |
| rejectChallengeInvite | Rejeita um convite de desafio. |

*Continua na página seguinte*

| Mutation | Descrição |
|---|---|
| deleteChallengeInvite | Elimina um convite de desafio. |
| createChallenge | Cria um novo desafio. |
| updateChallenge | Atualiza os dados de um desafio. |
| deleteChallenge | Remove um desafio existente. |
| startChallenge | Inicia um desafio. |
| endChallenge | Termina um desafio. |
| inviteFriend | Envia um convite de amizade a outro utilizador. |
| acceptFriendInvite | Aceita um convite de amizade. |
| rejectFriendInvite | Rejeita um convite de amizade. |
| deleteFriendInvite | Remove um convite de amizade. |

Para além do resumo geral apresentado nas tabelas anteriores, é importante destacar algumas mutations que apresentam lógica mais significativa. Abaixo encontram-se exemplos representativos que ilustram o tipo de operações que a API suporta e o seu impacto no funcionamento da aplicação.

A mutation `startChallenge` é responsável por iniciar oficialmente um desafio, sendo executada apenas pelo líder do desafio. Ao ser acionada, a operação verifica permissões, valida requisitos e inicializa os registos das conquistas associadas a cada participante.

- **Input:** `UserId`, `ChallengeId`
- **Output:** `MessageType` – objeto com os campos `successfull` e `message`
- **Entidades envolvidas:** `Users`, `UserChallenges`, `Challenges`, `ChallengeAchievements`, `UserChallengeAchievements`

**Resumo da Lógica:**

1. Verifica se o utilizador existe e pertence ao desafio indicado.
2. Garante que é o líder do desafio (único com permissão para iniciar).
3. Valida se o desafio possui pelo menos uma conquista.
4. Para cada utilizador inscrito no desafio:
    - Inicializa as conquistas com estado bloqueado (ou reinicia se já existirem).

5. Define a data de início do desafio com a hora atual.

6. Devolve uma mensagem de sucesso.

**Exemplo de resultado:**

```
{
  "successfull": true,
  "message": "CHALLENGE STARTED AND ACHIEVEMENTS INITIALIZED FOR ALL MEMBERS"
}
```

A mutation `lockUnlockChallengeAchievement` permite alternar o estado de uma conquista de desafio para um determinado utilizador, marcando-a como concluída ou não concluída. Esta alteração é refletida na percentagem de progresso do utilizador no desafio, e pode inclusive desencadear automaticamente o fim do desafio caso todos os objetivos tenham sido cumpridos.

- **Input:** `UserId`, `UserChallengeAchievementId`
- **Output:** `MessageType` (sucesso e mensagem textual)
- **Entidades envolvidas:** `Users`, `UserChallengeAchievements`, `Challenges`, `UserChallenges`

**Resumo da Lógica:**

1. Valida a existência do utilizador e da conquista.

2. Garante que a conquista pertence ao utilizador.

3. Alterna o estado da conquista entre bloqueado e desbloqueado.

4. Atualiza a data de desbloqueio caso tenha sido concluída.

5. Recalcula a média de progresso do utilizador (`AverageCompletion`) no desafio.

6. Se a média atingir 100% e o desafio ainda não tiver terminado, invoca automaticamente a função `endChallengeTransactional`.

**Encerramento Automático:** A função `endChallengeTransactional(challengeId)` é responsável por terminar oficialmente o desafio e atribuir os pontos aos utilizadores participantes com base no número de conquistas concluídas. Esta operação é executada dentro de uma transação de base de dados, garantindo integridade e consistência.

**Passos da função `endChallengeTransactional`:**

- Verifica a existência do desafio e se já foi terminado.
- Define a data de fim do desafio.
- Para cada utilizador participante:
  - Conta as conquistas concluídas.
  - Atualiza os pontos totais do utilizador com base nesse número.

**Exemplo de resultado:**

```
{
  "successfull": true,
  "message": "ACHIEVEMENT UNLOCKED!"
}
```

**inviteUserToChallenge** Esta mutation permite que um utilizador convide outro para participar num desafio existente. É verificado se o utilizador já faz parte do desafio e se existe algum convite pendente. Caso contrário, é criado um novo registo na entidade `ChallengeInvites` com o estado `PENDING`.

- **Input:** `UserId` (remetente), `ReceiverId` (destinatário), `ChallengeId`, `IsRequest`
- **Output:** `MessageType`
- **Entidades envolvidas:** `Users`, `Challenges`, `ChallengeInvites`

**Resumo da Lógica:**

1. Verifica se o remetente, destinatário e desafio existem.

2. Garante que o destinatário ainda não participa no desafio.
3. Impede convites duplicados (verifica se já existe um convite pendente).
4. Cria um novo registo em `ChallengeInvites` com a data atual e o estado `PENDING`.

**Exemplo de resultado:**

```
{
  "successfull": true,
  "message": "INVITE SENT SUCCESSFULLY"
}
```

**acceptChallengeInvite** : Esta mutation é responsável por aceitar um convite previamente enviado para participar num desafio. Após aceitar, o utilizador passa a fazer parte do desafio através da criação de um registo em `UserChallenges`.

- **Input:** `UserId`, `ChallengeId`
- **Output:** `MessageType`
- **Entidades envolvidas:** `Users`, `Challenges`, `ChallengeInvites`, `UserChallenges`

**Resumo da Lógica:**

1. Verifica a existência do utilizador e do desafio.
2. Garante que o utilizador ainda não participa no desafio.
3. Confirma a existência de convites pendentes para esse desafio.
4. Altera o estado dos convites para `ACCEPTED`.
5. Cria a relação entre utilizador e desafio na tabela `UserChallenges`.

**Exemplo de resultado:**

```
{
  "successfull": true,
  "message": "CHALLENGE ACCEPTED AND ACHIEVEMENTS CREATED SUCCESSFULLY"
}
```

## Upload e Armazenamento de Imagens

A aplicação permite o upload de imagens para serem utilizadas, por exemplo, como avatares dos utilizadores. Para tal, foi utilizado o middleware `multer`, que facilita o processamento de ficheiros enviados via HTTP.

O processo funciona da seguinte forma:

- É definido um diretório local chamado `Images` para armazenar as imagens recebidas. Se este diretório não existir no momento da execução, ele é criado automaticamente com o método `mkdirSync`.
- O `multer` é configurado para guardar os ficheiros numa pasta local: `Images`, atribuindo-lhes um nome único. Esse nome é composto por um prefixo com a data e hora atuais, seguido de um número aleatório, e do nome original do ficheiro, para evitar colisões e garantir unicidade.
- Através da rota `POST /Images`, o servidor aceita ficheiros submetidos por um campo chamado `file`, salvando-os no sistema de ficheiros local.
- As imagens armazenadas podem ser acedidas publicamente em: `/Images`, permitindo o acesso direto via URL.

Esta abordagem assegura a persistência dos ficheiros recebidos e facilita a sua utilização pela aplicação.

## Tratamento de Erros na API GraphQL

Ao contrário das APIs REST, que normalmente utilizam códigos de estado HTTP para indicar erros, o GraphQL adota um modelo diferente para o tratamento e comunicação de erros.

Quando uma operação falha, a resposta do servidor GraphQL inclui dois campos principais:

- **data**: contém os dados resultantes da operação, que podem estar completos, parciais ou até `null` se nenhum dado válido foi obtido.
- **errors**: um array de objetos que detalham os erros ocorridos durante a execução. Cada erro inclui uma mensagem descritiva, localização no documento GraphQL e, por vezes, códigos de erro adicionais.

Por exemplo, se a consulta para obter um utilizador falhar porque o utilizador não existe, a resposta pode ser semelhante a:

```
{
  "data": {
    "getUser": null
  },
  "errors": [
    {
      "message": "USER DOES NOT EXIST",
      "locations": [{ "line": 2, "column": 3 }],
      "path": ["getUser"],
      "extensions": {
        "code": "NOT_FOUND"
      }
    }
  ]
}
```

No backend, erros são lançados diretamente nos *resolvers* (por exemplo, `throw new Error("USER DOES NOT EXIST")`), sendo o Apollo Server responsável por capturar essas exceções e formatá-las conforme o GraphQL.

Além disso, para facilitar o feedback funcional, a API também utiliza um tipo personalizado `MessageType`, que permite retornar mensagens explícitas de sucesso ou falha nas mutações, complementando o mecanismo padrão de erros técnicos.

## Arquitetura Docker e Comunicação entre Containers

O ambiente da aplicação é composto por dois containers principais geridos através do `Docker Compose`:

- **Container** `mysql`: executa o servidor de base de dados MySQL versão 8. Este container expõe a porta interna 3306, que é mapeada para a porta 3307 da máquina host, permitindo acesso local para desenvolvimento e depuração. Os dados são persistidos num volume Docker nomeado `mysql-data`, garantindo que os dados permanecem intactos entre reinicializações do container.

- **Container** `backend`: contém a API Node.js que serve a aplicação GraphQL. É construído a partir de um `Dockerfile` dedicado (`Dockerfile.backend`) e expõe a porta 3000 para comunicação externa. Esta porta é mapeada diretamente para a porta 3000 do host.

### Comunicação entre os containers (Docker)

A comunicação entre o container `backend (API)` e o `mysql` é feita internamente na rede virtual criada pelo `Docker Compose`. O container do backend acede ao MySQL utilizando as variáveis de ambiente definidas no ficheiro `.env`, nomeadamente o hostname `mysql` (nome do serviço no `docker-compose.yml`), a porta 3306, e as credenciais de acesso.

Desta forma, o backend não depende da porta exposta para o host para se comunicar com a base de dados; ele usa a rede Docker interna, garantindo isolamento e maior segurança.

### Ordem de Inicialização e Volumes

No `docker-compose.yml`, o serviço `backend` depende do serviço `mysql`, o que assegura que o container da base de dados é iniciado primeiro. Além disso, os volumes persistentes garantem que os dados da base de dados (`mysql-data`) e os módulos Node.js (`node_modules`) são mantidos fora dos containers efémeros, preservando o estado entre reinicializações e reconstruções.

### Resumo do Fluxo

1. O `Docker Compose` cria uma rede isolada e lança os containers `mysql` e `backend`.

2. O container `backend` inicia a aplicação Node.js, que usa as variáveis ambiente para conectar-se ao `mysql` via rede Docker interna.

3. O MySQL responde às queries do backend na porta padrão 3306.

4. Para interações externas (ex. testes, desenvolvimento), o utilizador pode aceder à API pela porta 3000 do host, e à base de dados pela porta 3307 do host, conforme mapeado.

Este modelo facilita o desenvolvimento local, mantém os serviços separados e prepara o caminho para uma eventual implantação em ambientes de produção com configurações semelhantes.

## 4.1 Integração entre a Aplicação Android e a API GraphQL

Para permitir a comunicação entre a aplicação Android e a API desenvolvida em GraphQL, foi utilizada a biblioteca **Apollo Kotlin**, que facilita a execução de queries e mutations de forma segura e eficiente em aplicações Android escritas em Kotlin.

### 4.1.1 Obtenção do Schema com Rover

Antes de poder interagir com a API, é necessário obter o *schema* GraphQL. Esse schema define todas as operações possíveis (queries, mutations) e os tipos envolvidos. Para este projeto, utilizou-se o **Rover** — a ferramenta oficial da Apollo — para fazer introspeção do endpoint da API e gerar o ficheiro do schema:

Código 1: Comando para obter o schema GraphQL com Rover

```
rover subgraph introspect http://localhost:3000/graphql > schema.gr
```

Este comando guarda a definição completa da API no ficheiro `schema.graphql`, permitindo à aplicação conhecer a sua estrutura.

### 4.1.2 Geração Automática de Código com Apollo Codegen

Com o schema disponível, procedeu-se à configuração do **Apollo Codegen** para gerar automaticamente as classes necessárias a partir de ficheiros `.graphql` com as queries e mutations utilizadas pela aplicação.

Primeiro, adiciona-se a dependência ao `build.gradle.kts` (ou `build.gradle`) do módulo da aplicação:

Código 2: Configuração do Apollo Codegen no build.gradle.kts

```
apollo {
```

```
    service("service") {
        packageName.set("com.example.projecfinal.services.graphql")
        generateKotlinModels.set(true)
        schemaFile.set(file("src/main/graphql/service/schema.graphqls"))
        srcDir("src/main/graphql/service")
    }
}
```

As queries são colocadas em ficheiros `.graphql` dentro da pasta `src/main/graphql/service`, por exemplo:

Código 3: Exemplo de query em ficheiro .graphql

```
query GetAllUsers {
  getAllUsers {
    UserId
    UserName
    Email
    Biography
    MemberSince
    LastLogin
    TotalPoints
    AverageCompletion
    Image
  }
}
```

Depois de definir as queries, basta executar o seguinte comando para gerar automaticamente o código correspondente:

Código 4: Comando para gerar o código com Apollo Codegen

```
./gradlew generateApolloSources
```

### 4.1.3 Comunicação na Aplicação

No código da aplicação, a ligação ao endpoint da API é feita da seguinte forma:

Código 5: Inicialização do ApolloClient

```
val apolloClient = ApolloClient.Builder()
    .serverUrl("http://localhost:3000/graphql")
    .build()
```

Com o cliente configurado, pode-se invocar operações como a query `GetAllUsers`:

Código 6: Execução de uma query

```
val response = apolloClient.query(GetAllUsersQuery()).execute()
```

O diagrama seguinte ilustra todo o processo de integração entre a aplicação Android e a API GraphQL:

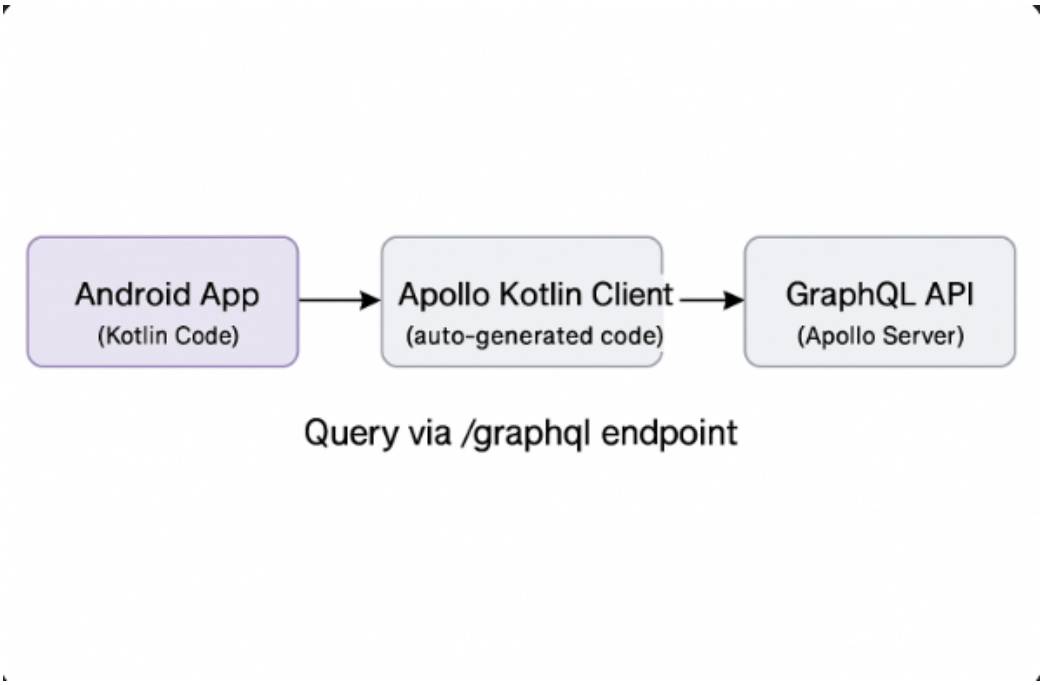


Figura 4.9: Query via /graphql Endpoint

### 4.1.4 Estrutura Modular de Integração com a API

De forma a garantir a modularidade e flexibilidade da aplicação, foi adotado um padrão de desenvolvimento baseado na definição de uma interface chamada *UserService*. Esta interface define todos os métodos necessários para a comunicação entre a aplicação e qualquer serviço externo, seja ele a API GraphQL desenvolvida neste projeto ou outro serviço.

A classe *DataSource* funciona como um contêiner ou ponto de abstração, sendo responsável por disponibilizar aos diferentes *ViewModels* os métodos definidos na interface *UserService*. Desta forma, os *ViewModels* não dependem diretamente da implementação concreta de um serviço, mas apenas da sua interface.

Por fim, foi criada uma classe *ServiceLocator* responsável por instanciar a implementação concreta do serviço que se pretende utilizar. No caso específico da nossa API GraphQL, a classe responsável por converter os dados recebidos para os tipos locais da aplicação desenvolvida em Kotlin é a *API-Service*. Esta classe implementa a interface *UserService*, encapsulando toda a lógica de comunicação com a API e garantindo que a aplicação permanece separada da tecnologia utilizada no backend.

# Capítulo 5

# Validação e Testes

De modo a vericar se o projeto estava a ser desenvolvido com sucesso foram-se fazendo testes, realizados por nos, estes consistiam em perceber que tipo de bugs este tinha, fossem eles de interacoes ao nivel da APP (frontend) ou mais logicas ao nivelo da API (backend) que nao funcionavam por completo ou de outras que funcionavam mas nao na sua totalidade. Estes testes alem de servirem para perceber certas funcionalidades como:

- Falta de informação retornada (API).
- Erros não ativarem quando deviam.
- etc.

Relativamente à APP, a maneira mais simples que se arranjou de testar a APP foi a utilização de Toast que é um método builtIn do Android Studio que permite mostrar na APP os erros que estão a acontecer. Mais tarde foram utilizados Logs para trabalharmos com Logcat, que é uma ferramenta do Android Studio que permite registos do sistema em tempo real.

Relativamente à API, o GraphQL oferece uma forma simples de testar as suas funcionalidades através do acesso direto ao endpoint: `https://IP:3000/graphql`. Para quem está habituado a testar aplicações REST, este ambiente funciona de forma semelhante ao Postman, mas adaptado a aplicações desenvolvidas com GraphQL.

A interface disponibilizada permite testar todas as queries e mutations definidas na API, num ambiente controlado e facil de utilizar.

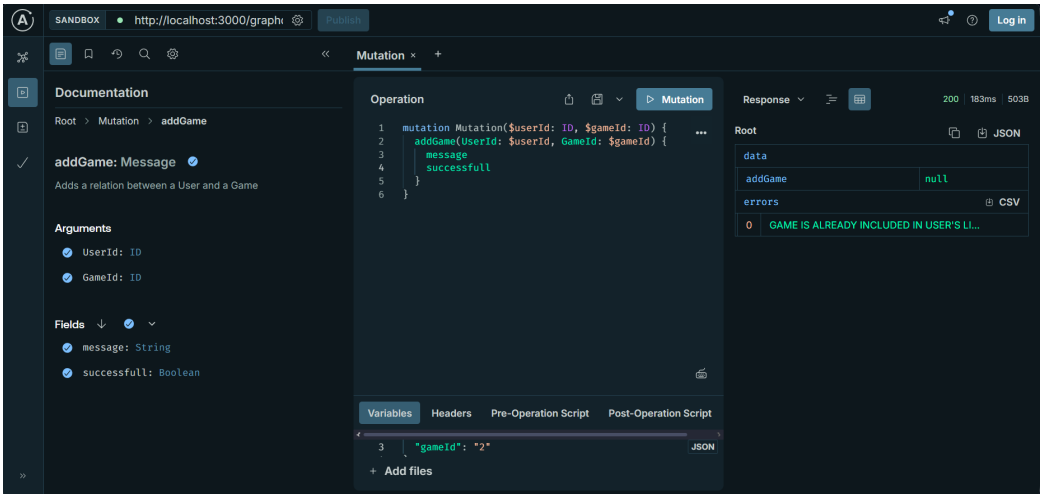


Figura 5.1: Exemplo de teste da função da mutation addGame

Do lado esquerdo, o programador pode construir a operação a ser testada, selecionando se se trata de uma query ou mutation, bem como definindo os parâmetros de entrada e de saída desejados.

Ao centro, encontra-se a estrutura em GraphQL da operação que está a ser executada, o que é bastante útil para garantir que a sintaxe está correta e que a operação está pronta a ser integrada com o frontend.

No lado direito, é possível observar a resposta gerada pela API, incluindo eventuais erros, o que facilita o processo de validação das funcionalidades desenvolvidas.

# Capítulo 6

# Conclusões e Trabalho Futuro

Chegando à parte conclusiva do projeto, pode-se afirmar que este consistiu no desenvolvimento, desde a raiz, de uma aplicação Android utilizando Android Studio, bem como de uma API desenvolvida com Node.js e GraphQL, responsável por toda a lógica de funcionamento da aplicação.

Ao longo do relatório, foram descritas as diversas etapas do desenvolvimento, permitindo acompanhar de forma clara todo o percurso até à criação do produto final.

Como trabalho Futuro podemos averiguar algumas funcionalidades que gostaríamos de ter implementado, mas por várias razões não foi possível como:

- Adicionar mais variedades de Challenges aos utilizadores. No nosso caso só é possível a partir do elemento Type no Challenge de fazer um tipo de challenge "EndFirst"que consiste em quem for o primeiro a terminar todos os ChallengeAchievements é o vencedor, mas gostaríamos também de adicionar, por exemplo, challenges por tempo.

- Adicionar um plugins, por exemplo, ao unity para que jogadores pudessem adicionar Achievements aos seus jogos criados. Isto era inicialmente a nossa ideia, mas devido à falta de tempo não foi possível ser relacionada, esta funcionalidade era bastante interessante, pois sem ela a 'app' está a mercê da honestidade dos jogadores para conquistar os Achievements em que competem.

Foi um grande desafio desenvolver algo completamente novo, recorrendo a tecnologias até então desconhecidas, como a construção de uma API e a

sua integração com a aplicação móvel. No entanto, a motivação de criar um projeto do zero — sem experiência prévia nessas tecnologias — tornou o processo extremamente enriquecedor e gratificante.

Olhando para o resultado, sentimos orgulho por termos conseguido concretizar um sistema funcional, completo e com uma arquitetura bem definida, representando uma evolução significativa das nossas competências técnicas.

# Apêndice A

# Modelo Entidade Associação (EA) da APP

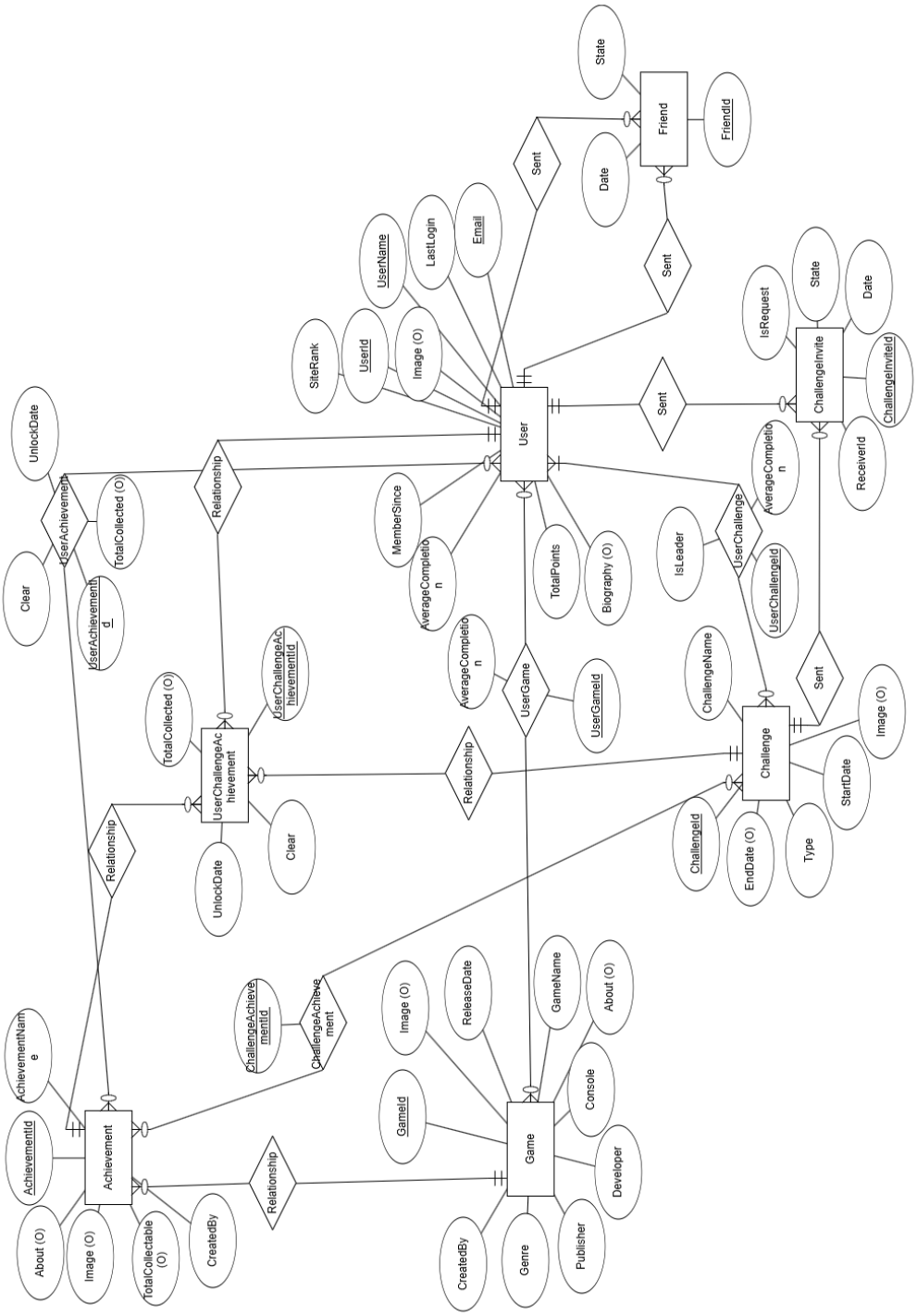


Figura A.1: Diagrama de Entidade e Associação

# Apêndice B

# Modelo Relacional da APP

Escrever aqui o detalhe adicional que melhor explique outro aspecto (diferente do que está no apêndice A) descrito no corpo principal do documento ...

# Bibliografia

[Achievements, 2012] Achievements, R. (2012). Retro achievements. `https://retroachievements.org`.

[Adventures, 2004] Adventures, J. C. (2004). *Jackie Chan Adventures*. Ratchet and Clank. Sony Computer Entertainment.

[Arch, 2012] Arch, R. (2012). Retro arch. `https://www.retroarch.com`.

[Cooper e the Thievius Raccoonus, 2004] Cooper, S. e the Thievius Raccoonus (2004). *Sly Cooper and the Thievius Raccoonus*. Sly Cooper. Sony Computer Entertainment.

[Ratchet e Clank, 2004] Ratchet e Clank (2004). *Ratchet and Clank*. Ratchet and Clank. Playstation.

[Yellow, 1998] Yellow, P. (1998). *Pokemon Yellow*. Pokemon. Nintendo.