Stahp!

Um Stop (sem o Stop)

O projeto desenvolvido é uma implementação de um jogo com o objetivo similar ao "**Stop!**" popular, com a utilização de arquitetura **cliente-servidor**, utilizando uma **REST** para a definição de sua API.

Regras do Jogo

Uma partida do jogo pode ter **um ou mais participantes**. Assim que a partida é **criada por um jogador** ela fica disponível para que outros jogadores **se juntem** à partida, até que o criado decida **iniciar a partida**.
No início da partida, é selecionada uma lista de pares **<inicial, tema>**, que chamaremos de **desafios**. Ao iniciar sua jogada, cada participante terá que um determinado **limite de tempo** para responder quantos desafios conseguir.
Para cada desafio, a **resposta do jogador** é comparada a uma lista de **respostas possíveis** cadastradas no sistema e a ela é atríbuido um valor de **recompensa**. A **pontuação final** de um jogador é a **soma das recompensas** obtidas por ele.

Ficam como **parâmetros de implementação** a seleção de temas, o limite de tempo e o cálculo de recompensas. Esses podem ser alterados conforme estudos para obter melhor de jogabilidade.

Protocolo Utilizado

O protocolo utilizado para a comunicação entre o cliente e o servidor é uma **API** utilizando HTTP baseada no conceito **REST** (Representational state transfer). De forma geral, REST é um estilo de desenvolvimento de APIs que tem como um dos principais conceitos o uso de **requisições que não mantém sessão** no servidor.
Assim, **toda a informação** referente a uma mensagem do cliente ao servidor **é enviada na requisição**. Isso permite que os clientes interaja diversos servidores de forma transparente, como se utilizassem um único servidor.
A API desenvolvida segue também alguns conceitos de **RESTful**, especialmente sobre o **uso semântico dos métodos HTTP** (GET para obter dados, POST para enviar informações, etc) e o a produção de dados em formatos estruturados, como **JSON** e **XML**.

Definição da API

A API implementada está descrita nas seções a seguir, com os **tipos de dados (entidades)** utilizados, as respectivas URLs e os métodos HTTP disponíveis para cada uma, além de descrições referentes.

Para todos os métodos, exceto a criação e obtenção de um perfil de jogador, é necessário realizar **autenticação, feita por uma chave de acesso** passada como parâmetro da requisição.

Entidades

- **PlayerEntity**: perfil de um jogador
    - id (String)
    - name (String)
- **MatchEntity**: dados de uma partida
    - id (String)
    - status (String) - valores possíveis: CREATED, STARTED, FINISHED
    - created (Date)
    - updated (Date)
    - creator (PlayerEntity)
    - timeLimit (Integer)
    - entries (EntryEntity[])
    - responded (boolean) - jogador autorizado já jogou?
    - joined (boolean) - jogador autorizado se juntou à partida?
- **EntryEntity**: entrada de um jogador numa partida
    - words (String[]) - palavras enviadas em resposta aos desafios
    - player (PlayerEntity)
    - score (Integer)
- **ChallengeEntity**: descrição de um desafio
    - topic (String)
    - initial (String)

Métodos

- **/players**
    - **POST:** cria um novo perfil de jogador. Retorna o *PlayerEntity* criado
- **/players/*{id}***
    - **GET:** obtém o *PlayerEntity* do jogador para o *{id}* dado
- **/matches**
    - **GET:** obtem a lista de *MatchEntity* do jogador autorizado
    - **POST:** cria uma nova partida. Retorna o *MatchEntity* criado
- **/matches/all**
    - **GET:** obtem a lista de *MatchEntity*
- **/matches/{id}**
    - **GET:** obtem o *MatchEntity* para a partida identificada por *{id}*

    - **POST:** caso o jogador autorizado seja o criador da partida, inicia a partida; caso contrário, adiciona o jogador à lista de participantes da partida. Retorna a *MatchEntity* modificado
- **/matches/{id}/words**
    - **GET:** obtém a lista de *ChallengeEntity* para a partida
    - **POST:** salva a lista de respostas do jogador autorizado. Retorna a *EntryEntity* resultante

Implementação

Servidor

O servidor foi desenvolvido em Java, utilizando **JAX-RS** (implementação do framework **Jersey**) para a definição da interface RESTful e a camada de persinstência utiliza **JPA** (implementação do **Hibernate ORM**). É utilizado o módulo de **injeção de depêndencias do Spring**, além de outras bibliotecas como **Log4j** (gerenciamento de logs) e o **Grizzly** (servidor embarcado).
A arquitetura do projeto tenta seguir os conceitos de uso de service e DAO, além da utilização de um controller para a lógica do jogo.

Cliente

O cliente foi implementado como um **aplicativo para Android**, utilizando uma interface gráfica básica, seguindo o estilo Holo. A biblioteca **Volley** (desenvolvida pelo Google para uso no aplicativo da Play Store) foi utilizada para o gerenciamento das requisições HTTP. Outras bibliotecas também foram usadas, como **Crouton** (mensagens de notificação) e **gson** (biblioteca do Google para parsing de JSON utilizando reflection).
Foi desenvolvido uma **camada de acesso à API do servidor**, que centraliza a construção das requests e seu envio.

Observações, falhas e possíveis melhorias

O **modelo de dados** utilizado não está maduro, especialmente no uso da definição dele pelo ORM. Com mais atenção a isso poderíamos melhorar a eficiência do servidor. O mesmo se aplica ao **protocolo da API**.
A **organização do projeto** para Android está bem fraca, uma refatoração geral tornaria ele bem mais compreensível. A **interface de usuário** do aplicativo também poderia se tornar mais amigável.
O **calculo das recompensas** está simplista, o uso de algoritmos mais refinados, como o cálculo da **distância de Levenshtein** poderia dar resultados mais justos para erros de grafia. Métodos mais complexos para a geração da **lista de desafios** também seriam desejáveis.