CIÊNCIA DA COMPUTAÇÃO
PROGRAMAÇÃO ORIENTADA A OBJETOS

TRABALHO FINAL FASE 1

Gabriel Araújo Velasco
João Pedro de Oliveira Martins Vieira
Kaio Augusto de Souza
Victor Hugo Ignacio Franco

Uberlândia

Junho de 2022

Sumário

1. Burocracias

1.1. Participantes

Kaio Augusto de Souza - 11921BCC040
João Pedro de Oliveira Martins Vieira - 11921BCC017
Victor Hugo Ignacio Franco - 11921BCC020
Gabriel Araújo Velasco - 11921BCC003

1.2. Problema escolhido

App para um jogo de Perguntas e Respostas (quiz)

2. Subproblemas

2.1. Representação do quiz

O primeiro problema que encontramos ao reproduzir o jogo, foi em como iríamos de fato representar o próprio quiz, depois de pensarmos bastante escolhemos fazer um aplicativo web, com a base do jogo funcionando utilizando Java com a framework Spring e para o frontend provavelmente iremos utilizar React ou Angular, mas ainda estamos em aberto com relação a isso, optamos por fazer assim ao invés de utilizar Swing por estar simulando um ambiente mais real.

3. Dependências e bibliotecas

Dependências que utilizamos até o momento, no resultado final podem ter mais ou menos dependências

- Spring Boot Starter Data JPA;
- Spring Boot Starter WEB;
- Spring Boot Starter Security;
- Postgresql;
- Lombok;

4. Classes

Aqui será apresentado um pouco das classes que temos em mente para realizar o projeto.

4.1. Entities

Aqui temos o modelo de nossos objetos

4.1.0. Question

Para a entidade “Question” ou questão, nós temos as seguintes variáveis:

- ID: Serve como um identificador único para essa questão
- Question: A questão em si em forma de texto
- OptionA, B, C, D: Aqui temos todas as 4 opções de resposta para a questão;
- CorrectAnswer: A questão correta
- Difficulty: O nível de dificuldade da questão. Ver mais em 4.2.1
- Category: A categoria da pergunta. Ver mais em 4.2.0

```
@Entity
@Table(name = "questions")
public class Question {

    @Id
    @GeneratedValue
    private UUID id;

    private String question;

    private String optionA, optionB, optionC, optionD;

    private String correctAnswer;

    @Enumerated(EnumType.STRING)
    private Difficulty difficulty;

    @Enumerated(EnumType.STRING)
    private Category category;
}
```

4.1.1. User

Para a entidade “User” utilizamos a seguintes variáveis:

- ID: Serve como um identificador único deste usuário
- Name: Nome do usuário
- Email: Email do usuário
- Password: Senha do usuário
- Roles: “Cargos” do usuário, basicamente as permissões do usuário permitindo adicionar ou deletar perguntas por exemplo

```
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue
    private UUID id;

    private String name;

    @Column(unique = true)
    private String email;

    private String password;

    @ManyToMany(fetch = FetchType.EAGER)
    private List<Role> roles;
}
```

4.1.2. Role

Para a entidade “Role” ou cargos utilizamos as seguintes variáveis:

- ID: Serve como um identificador único deste cargo
- Name: Nome do cargo

```
@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue
    private UUID id;

    private String name;
}
```

4.2. Enums

Utilizamos aqui enums para definir constantes em nosso programa

4.2.0. Category

A Enum categoria contém as categorias predefinidas de perguntas que temos.

```
public enum Category {

    MATH("Matemática"),
    ENGLISH("Inglês"),
    SCIENCE("Ciências"),
    HISTORY("História"),
    GEOGRAPHY("Geografia"),
    PHYSICS("Física"),
    BIOLOGY("Biologia"),
    MISCELLANEOUS("Outros");

    @Getter
    private final String portugueseName;

    Category(String portugueseName) {
        this.portugueseName = portugueseName;
    }
}
```

4.2.1. Difficulty

A enum difficulty possui o nível de dificuldade das perguntas.

```
public enum Difficulty {
    EASY,
    MEDIUM,
    HARD
}
```

4.3. Repositories

As classes repositórios no Spring JPA serve como a nossa ponte entre o aplicativo e o banco de dados (Postgresql)

4.3.1. User Repository

Como um exemplo de classe repositório já temos pronta a UserRepository

```
@Repository
public interface UserRepository extends JpaRepository<User, UUID> {

    @Query("SELECT user FROM User user WHERE user.email = :email")
    Optional<User> findByEmail(String email);
}
```

4.4. Services

As classes services no Spring funcionam como a abstração das classes repositório, onde definimos as regras de serviço para salvar, deletar, atualizar, etc… as classes repositório.

4.4.0. User Repository

Como exemplo de uma função da classe UserRepository temos a função de salvar

```
public UserDTO save(UserDTO userDTO) {
    Optional<User> user = userRepository.findByEmail(userDTO.getEmail());

    if (user.isPresent()) return null;

    User newUser = User.builder()
            .name(userDTO.getName())
            .email(userDTO.getEmail())
            .password(passwordEncoder.encode(userDTO.getPassword()))
            .roles(userDTO.getRoles())
            .build();

    this.userRepository.save(newUser);

    return new UserDTO(newUser);
}
```

4.5. Controllers

As classes controller funcionam como a nossa resposta final para a API REST, lá retornamos todas as respostas que queremos para nossa API.

4.5.0. User Controller

Um exemplo seria novamente a função de salvar um usuário:

```
@PostMapping
public ResponseEntity<Object> save(@RequestBody UserDTO userDTO) {
    UserDTO user = this.userService.save(userDTO);

    if (user == null) {
        return ResponseEntity.status(HttpStatus.CONFLICT).body("User already
exists");
    }

    return ResponseEntity.status(HttpStatus.CREATED).body(user);
}
```