Vem Ser DBC

TEXT GAME

BATTLE OF WORLDS

EQUIPE HELLFIRE CLUB

ROUND 3.3

PLAYERS

Eric de Sousa

Gustavo Teichmann

Paulo Ricardo

ROUND 3.3

No Módulo 1 desenvolvemos um jogo RPG em formato de texto onde o usuário pode interagir com um vilão através de dinâmicas de ataque e defesa.

O desafio do Módulo 2 foi implementar na aplicação o acesso ao banco de dados e no modulo 3, implementamos API do game.

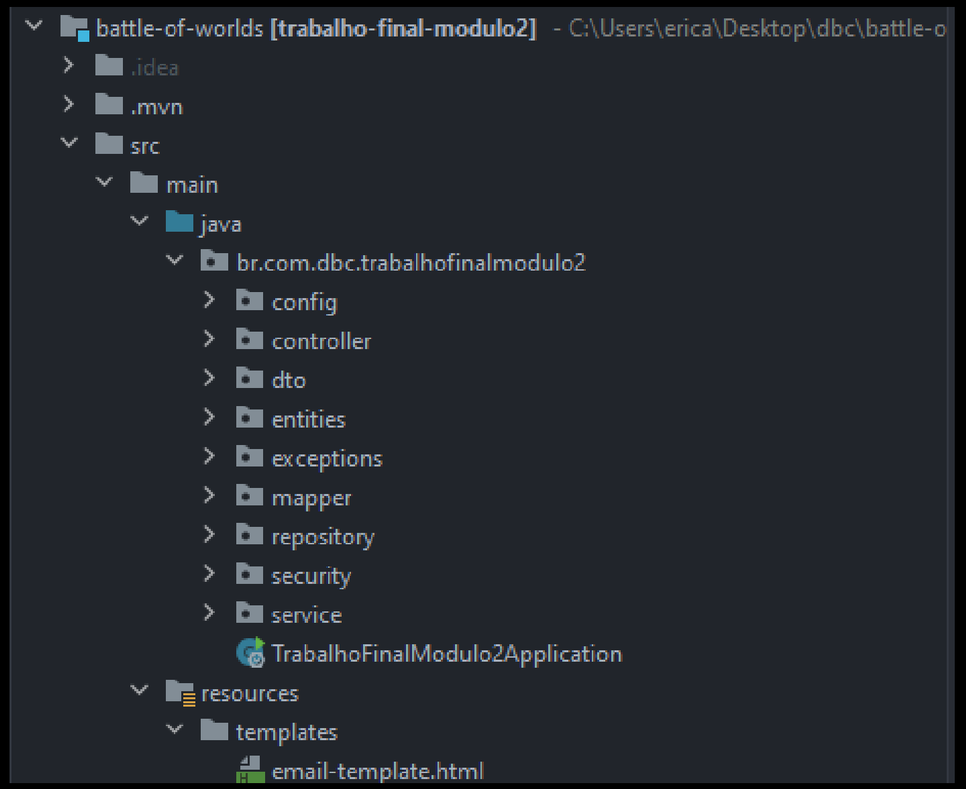
O desafio no Módulo 3.2 criamos a ligação do banco através do JPA.

Agora no Módulo 3.3 implementamos a parte de segurança da aplicação

VEM SER DBC

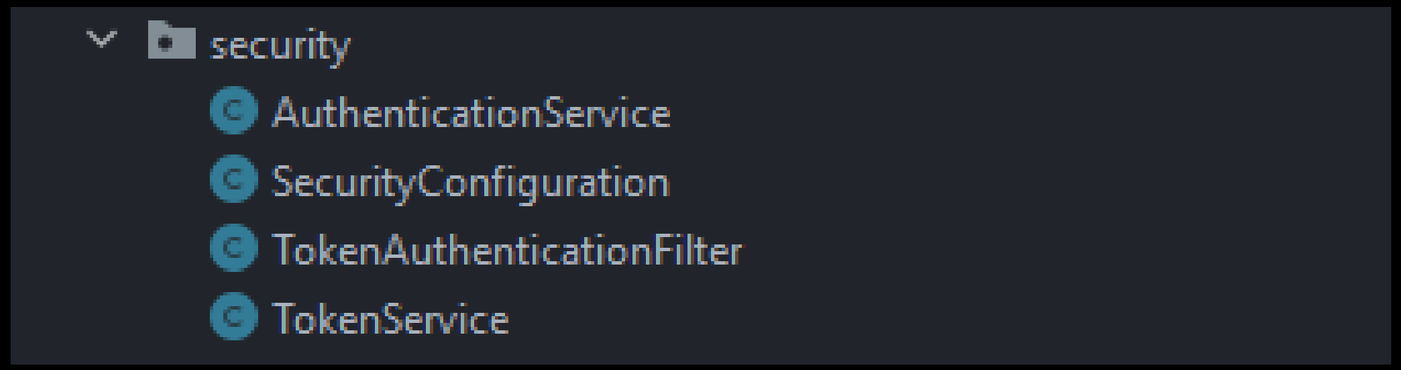
PADRÃO MVC

MODEL VIEW CONTROLLER



VEM SER DBC

CRIAÇÃO DAS CLASSES PARA SECURITY



AuthenticationService para fazer a parte de autenticação do sistema.

SecurityConfiguration responsável de passarmos quem pode acessar e o que pode acessar.

TokenAuthenticationFilter filtro para autenticação com o nosso token.

TokenService responsavel pela criação e validação do Token.

LIGAÇÃO DE TABELAS JOGADOR E CARGOS

```java
@Getter
@Setter
@Entity(name = "cargo")
public class CargoEntity implements GrantedAuthority {   Andrade, 28/07/2022 22:45 • [Feature] Entitys
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "cargo_seq")
    @SequenceGenerator(name = "cargo_seq", sequenceName = "seq_cargo", allocationSize = 1)
    @Column(name = "id_cargo", nullable = false)
    private Integer idCargo;

    @Column(name = "nome", nullable = false)
    private String nome;

    @JsonIgnore
    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "jogador_cargo",
            joinColumns = @JoinColumn(name = "id_cargo"),
            inverseJoinColumns = @JoinColumn(name = "id_jogador"))
    private Set<JogadorEntity> jogadores;


    @Override
    public String getAuthority() { return this.nome; }
}
```

Implementamos a criação de roles cargos para limitar os acessos por função de cada jogador

REFATORAÇÃO DA CLASSE JOGADOR

```java
public class JogadorEntity implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "jogador_seq")
    @SequenceGenerator(name = "jogador_seq", sequenceName = "seq_jogador", allocationSize = 1)
    @Column(name = "id_jogador", nullable = false)
    private Integer idJogador;

    @Column(name = "nome_jogador", nullable = false)
    private String nomeJogador;

    @Column(name = "senha", nullable = false)
    private String senha;

    @JsonIgnore
    @OneToMany(mappedBy = "jogadorEntity")
    private Set<PersonagemEntity> personagems;

    @Column(name = "email", nullable = false)
    private String email;

    @JsonIgnore
    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "jogador_cargo",
            joinColumns = @JoinColumn(name = "id_jogador"),
            inverseJoinColumns = @JoinColumn(name = "id_cargo"))
    private Set<CargoEntity> cargos;
    Andrade, 28/07/2022 22:45 • [Feature] Entitys
    @Column(name = "enable", nullable = false)
    private Boolean enable;
```

Refatoramos a Classe de Jogador onde implementamos o UserDetails e fizemos a ligação ManyToMany com a tabela Cargo

UTILIZAÇÃO DA CRIPTOGRAFIA SCRYPTPASSWORDENCODER

```java
public LoginRetornaDTO criarUsuario(LoginCreateDTO loginCreateDTO){
    SCryptPasswordEncoder sCryptPasswordEncoder = new SCryptPasswordEncoder();
    String senhaUsuario = loginCreateDTO.getSenha();
    loginCreateDTO.setSenha(sCryptPasswordEncoder.encode(senhaUsuario));
    JogadorEntity jogadorEntity = jogadorMapper.fromCreateLoginDTO(loginCreateDTO);
    jogadorRepository.save(jogadorEntity);
    LoginRetornaDTO loginRetornaDTO = jogadorMapper.toLoginDTO(jogadorEntity);
    return loginRetornaDTO;
}
```

Fizemos a ligação com o SCryptpasswordEncoder

SCRYPTPASSWORDENCODER

$e0801$G+2Fod2V+iv7b9xtKmliqW9ld+b3BAPtfuZj8vgjTwRA5eMAgUkZEg3MsSzGaa8DDLIaSFvRx+ATHOsF8TOpbQ==$kQ2WULY6lH535Ft2sO++Kd93sA9uYEtMFuiVnNU3XBU=

Nossa criptografia e dividida em configurações de algoritmo + Salt + Hash, onde cada $ separa cada um dos elementos

VAMOS CONFERIR A API FUNCIONANDO?

VEM SER DBC