

ROTEIRO - MIGRAR AS ENTIDADES DESAFIOS E PARTIDAS PARA O CONTEXTO BASEADO EM MICROSERVICES

Neste roteiro apresento as atividades necessárias para que possa migrar as entidades desafios e partidas para o contexto baseado em microservices, onde são contempladas adequações tanto no API Gateway quanto a criação do microservice desafios.

A intenção é que este material sirva como um guia, de modo que você possa conduzir essa atividade, antes de visualizar o vídeo em que apresento o resultado da minha implementação.

- 1- Começando com o **refactoring** nas **entidades categorias e jogadores** que pertencem ao microservice admin-backend

a. API Gateway

- i. Atualizar/Criar as interfaces Categoria e Jogador baseado nas definições atualizadas dos modelos em JSON

```
jogador.interface.ts 241 Bytes

1  export interface Jogador {
2      readonly _id: string;
3      readonly telefoneCelular: string;
4      readonly email: string;
5      categoria: string;
6      nome: string;
7      ranking: string;
8      posicaoRanking: number;
9      urlFotoJogador: string;
10 }
```

```
categoria.interface.ts 225 Bytes

1  export interface Categoria {
2      readonly _id: string;
3      readonly categoria: string;
4      descricao: string;
5      eventos: Array<Evento>;
6  }
7
8
9  interface Evento {
10     nome: string;
11     operacao: string;
12     valor: number;
13 }
```

b. Microservice admin-backend

- i. Passe a utilizar variáveis de ambiente, com o ConfigModule
- ii. Na classe JogadoresService, será necessário deixar de popular categoria nos métodos consultarTodosJogadores e consultarJogadorPeloId. A partir de agora retornaremos apenas o _id.

```

27
28
29     async consultarTodosJogadores(): Promise<Jogador[]> {
30         try {
31             return await this.jogadorModel.find().exec()
32         }
33         catch(error) {
34             this.logger.error(`error: ${JSON.stringify(error.message)}`)
35             throw new RpcException(error.message)
36         }
37     }

```

2- Adequações no API Gateway

- a. Modularize sua aplicação
 - i. **Crie o módulo desafios** (Module e Controller). Neste momento crie apenas a estrutura padrão gerada pelo Nest
- b. Agora realize o processo de **migração** da **entidade desafios** em conjunto com a **entidade partidas**:

- i. Realize a **migração** dos **Dtos**

Como nosso modelo sofreu atualizações importantes irei compartilhar as definições que utilizei.

1. criar-desafio.dto.ts

```

criar-desafio.dto.ts 418 Bytes
Edit Web IDE

1  import { IsNotEmpty, IsArray, ArrayMinSize, ArrayMaxSize, IsDateString } from 'class-validator';
2  import { Jogador } from '../../jogadores/interfaces/jogador.interface';
3
4  export class CriarDesafioDto {
5      @IsNotEmpty()
6      @IsDateString()
7      dataHoraDesafio: Date;
8
9      @IsNotEmpty()
10     solicitante: string;
11
12     @IsNotEmpty()
13     categoria: string;
14
15     @IsArray()
16     @ArrayMinSize(2)
17     @ArrayMaxSize(2)
18     jogadores: Jogador[]
19
20
21 }

```

2. atualizar-desafio.dto.ts

```

atualizar-desafio.dto.ts 183 Bytes
Edit Web IDE

1  import { DesafioStatus } from '../desafio-status.enum';
2  import { IsOptional } from 'class-validator';
3
4  export class AtualizarDesafioDto {
5
6      @IsOptional()
7      status: DesafioStatus;
8
9  }

```

3. atribuir-desafio-partida.dto.ts

```
atribuir-desafio-partida.dto.ts 304 Bytes Edit Web IDE

1 import { IsNotEmpty } from 'class-validator';
2 import { Resultado } from '../interfaces/partida.interface';
3 import { Jogador } from '../jogadores/interfaces/jogador.interface';
4
5 export class AtribuirDesafioPartidaDto {
6
7     @IsNotEmpty()
8     def: Jogador
9
10    @IsNotEmpty()
11    resultado: Array<Resultado>
12
13 }
```

ii. Realize a **migração** das **interfaces**

Como nosso modelo sofreu atualizações importantes irei compartilhar as definições que utilizei.

1. desafio.interface.ts

```
desafio.interface.ts 370 Bytes Edit Web IDE

1 import { Jogador } from '../jogadores/interfaces/jogador.interface';
2 import { DesafioStatus } from '../desafio-status.enum';
3
4 export interface Desafio {
5
6     dataHoraDesafio: Date
7     status: DesafioStatus
8     dataHoraSolicitacao: Date
9     dataHoraResposta: Date
10    solicitante: Jogador
11    categoria: string
12    partida?: string
13    jogadores: Array<Jogador>
14
15 }
```

2. partida.interface.ts

```
partida.interface.ts 266 Bytes Edit Web IDE

1 import { Jogador } from '../jogadores/interfaces/jogador.interface';
2
3 export interface Partida {
4     categoria?: string
5     desafio?: string
6     jogadores?: Jogador[]
7     def?: Jogador
8     resultado?: Resultado[]
9 }
10
11 export interface Resultado {
12     set: string
13 }
```

iii. Realize a migração do **desafio-status.enum.ts**

iv. Realize a migração do **desafio-status-validation.pipe.ts**

- v. Atualize o **client-proxy.ts**, criando um novo método que será responsável por lidar com a nova fila desafios.

- vi. Na **classe DesafiosController**, lembre-se de criar um novo ClientProxy responsável por lidar apenas com tópicos do microservice desafios.

```
21      /*
22         Criamos um proxy específico para lidar com o microservice
23         desafios
24      */
25      private ClientDesafios =
26      this.clientProxySmartRanking.getClientProxyDesafiosInstance()
27
28      private ClientAdminBackend =
29      this.clientProxySmartRanking.getClientProxyAdminBackendInstance()
30
```

- vii. Na **classe DesafiosController**, realize a **migração dos métodos**:

1. Método **criarDesafio**

Encaminhe um novo desafio para o message broker, considerando o uso de um event emitter.

[Desafio] Antes de encaminhar um novo desafio para o message broker, realize as seguintes validações:

- Verifique se os jogadores que fazem parte do desafio realmente estão cadastrados
- Verifique se os jogadores realmente pertencem à categoria que foi informada no desafio
- Verifique se o solicitante do desafio é um jogador da partida
- Verifique se a categoria informada realmente está cadastrada

2. Método **consultarDesafios**

Atenção ao uso do padrão requestor/responder

[Desafio] Antes de consultar os desafios de um jogador, verifique se o jogador realmente está cadastrado.

3. Método **atualizarDesafio**

Encaminhe o Dto para o message broker, considerando o uso de um event emitter.

[Desafio] Antes de prosseguir com a atualização de um desafio, realize as seguintes validações:

- Verifique se o desafio informado realmente está cadastrado
- Lembre-se de que somente desafios com status PENDENTE, podem ser atualizados

4. Método **deletarDesafio**

[Desafio] Antes de prosseguir com a deleção lógica de um desafio, verifique se o desafio existe.

5. Método **atribuirDesafioPartida**

[Desafio] Antes de acionar um tópico específico para criação da partida, lembre-se de realizar as seguintes validações:

- Verifique se o desafio informado realmente está cadastrado
- Recuse com mensagem específica requisições que tentem atualizar desafios com status REALIZADO
- Lembre-se de que somente desafios com status ACEITO podem receber uma partida
- Verifique se o jogador vencedor realmente faz parte do desafio

3- Agora vamos para nosso **microservice desafios**

- a. Crie um **novo projeto**
nest new micro-desafios
- b. Instale as **dependências**
npm install @nestjs/microservices
npm install amqplib amqp-connection-manager
npm install @nestjs/mongoose mongoose
npm install --dev @types/mongoose
npm install @nestjs/config
- c. Modularize sua aplicação
 - i. **Crie os módulos desafios, partidas** (Module, Service e Controller). Também **crie** o módulo **proxymq** (Module). Neste momento crie apenas a estrutura padrão gerada pelo Nest
- d. Utilize variáveis de ambiente, com o ConfigModule
- e. Lembre-se de que este **microservice** irá interagir com a **fila desafios** no RabbitMQ
- f. No arquivo **app.module.ts**, lembre-se de utilizar um **database específico**: srdesafios
- g. No **módulo desafios**:
 - i. Realize a **migração** das **interfaces** e do **schema**
Como nosso modelo sofreu atualizações importantes irei compartilhar as definições que utilizei.
 1. desafio.interface.ts

```
desafio.interface.ts 345 Bytes  Edit  Web IDE

1  import { Document } from 'mongoose';
2  import { DesafioStatus } from '../desafio-status.enum'
3
4  export interface Desafio extends Document {
5
6      dataHoraDesafio: Date
7      status: DesafioStatus
8      dataHoraSolicitacao: Date
9      dataHoraResposta?: Date
10     solicitante: string
11     categoria: string
12     partida?: string
13     jogadores: string[]
14
15 }
```

2. desafio.schema.ts

```
desafio.schema.ts 701 Bytes
1 import * as mongoose from 'mongoose';
2
3 export const DesafioSchema = new mongoose.Schema({
4   dataHoraDesafio: { type: Date },
5   status: { type: String },
6   dataHoraSolicitacao: { type: Date },
7   dataHoraResposta: { type: Date },
8   //solicitante: {type: mongoose.Schema.Types.ObjectId, ref: "Jogador"},
9   solicitante: {type: mongoose.Schema.Types.ObjectId},
10  //categoria: {type: String },
11  categoria: {type: mongoose.Schema.Types.ObjectId},
12  jogadores: [{
13    type: mongoose.Schema.Types.ObjectId,
14    //ref: "Jogador"
15  }],
16  partida: {
17    type: mongoose.Schema.Types.ObjectId,
18    ref: "Partida"
19  },
20 }, {timestamps: true, collection: 'desafios' })
21
22
23
```

- ii. Lembre-se de migrar o desafio-status.enum.ts
- iii. **Migrando** os métodos da classe **DesafiosService**
 - 1. Método **criarDesafio**
 - 2. Método **consultarTodosDesafios**
 - 3. Método **consultarDesafiosDeUmJogador**
 - 4. Método **consultarDesafioPeloId**
Novo método. Realiza a consulta de um desafio pelo seu id.
 - 5. Método **atualizar desafio**
 - 6. Método **atualizarDesafioPartida**
Este método sofreu alterações e será responsável por atribuir o id de uma nova partida a um desafio.
 - 7. Método **deletarDesafio**

Observação: Lembre-se de envolver seus métodos com o bloco try/catch, lançando uma RpcException, em caso de erro.
- iv. **Migrando** a classe **DesafiosModule**
Lembre-se de importar o MongooseModule, registrando o DesafioSchema
- v. **Migrando** a classe **DesafiosController**
 - 1. Método **criarDesafio**
Utilize um event subscriber para recuperar as mensagens que estão no RabbitMQ
 - 2. Método **consultarDesafios**
Utilize um responder para retornar um payload para o cliente
[Desafio] Lembre-se de que esse método será responsável por acionar três diferentes métodos do provider DesafiosService:
 - consultarDesafiosDeUmJogador
 - consultarDesafioPeloId
 - consultarTodosDesafios
Ou seja, você deverá avaliar em que momento estes diferentes métodos serão acionados. Lembre-se que você deverá estar bem alinhado com o API Gateway.
 - 3. Método **atualizarDesafio**

Recupere o id do desafio e o Dto e realize a atualização do desafio

4. Método **deletarDesafio**

5. Método **atualizarDesafioPartida**

Este método contempla uma nova implementação. Este tópico deverá ser acionado pela classe PartidasService, após a persistência de uma nova partida, e será responsável por atribuir o id da partida ao desafio.

```
77 @EventPattern('atualizar-desafio-partida')
78 async atualizarDesafioPartida(
79   @Payload() data: any,
80   @Ctx() context: RmqContext
81 ) {
82   const channel = context.getChannelRef()
83   const originalMsg = context.getMessage()
84   try {
85     this.logger.log(`idPartida: ${data}`)
86     const idPartida: string = data.idPartida
87     const desafio: Desafio = data.desafio
88     await this.desafiosService.atualizarDesafioPartida(idPartida, desafio)
89     await channel.ack(originalMsg)
90   } catch(error) {
91     const filterAckError = ackErrors.filter(
92       ackError => error.message.includes(ackError)
93     )
94     if (filterAckError) {
95       await channel.ack(originalMsg)
96     }
97   }
98 }
```

Observações:

- Não se esqueça de envolver seus métodos com o bloco try/catch/finally
- Não se esqueça de aplicar o acknowledge nas mensagens.

h. No **módulo partidas**:

Este é um novo módulo. Em nossa implementação anterior não existia um módulo específico para partidas.


i. Realize a **migração** das **interfaces** e do **schema**

Como nosso modelo sofreu atualizações importantes irei compartilhar as definições que utilizei.

1. partida.interface.ts

```
partida.interface.ts 246 Bytes
1 import { Document } from 'mongoose';
2
3 export interface Partida extends Document{
4   categoria: string
5   desafio: string
6   jogadores: string[]
7   def: string
8   resultado: Array<Resultado>
9 }
10
11 export interface Resultado {
12   set: string
13 }
```

2. partida.schema.ts

```
partida.schema.ts 435 Bytes  Edit Web IDE  
1 import * as mongoose from 'mongoose';  
2  
3 export const PartidaSchema = new mongoose.Schema({  
4  
5   desafio: { type: mongoose.Schema.Types.ObjectId },  
6   categoria: {type: mongoose.Schema.Types.ObjectId},  
7   jogadores: [{  
8     type: mongoose.Schema.Types.ObjectId,  
9   }],  
10  def: { type: mongoose.Schema.Types.ObjectId },  
11  resultado: [  
12    { set: {type: String} }  
13  ]  
14  
15  }, {timestamps: true, collection: 'partidas' })
```

ii. Criar o método criarPartida da classe PartidasService

Aqui temos uma nova implementação. Este provider, além de realizar a persistência de uma nova partida no banco de dados, também será um event emitter. Deste modo, ele irá promover a atualização de um desafio, através do tópico atualizar-desafio-partida.

```
22 async criarPartida(partida: Partida): Promise<Partida> {  
23   try {  
24     /*  
25      Iremos persistir a partida e logo em seguida atualizaremos o  
26      desafio. O desafio irá receber o ID da partida e seu status  
27      será modificado para REALIZADO.  
28     */  
29     const partidaCriada = new this.partidaModel(partida)  
30     this.logger.log(`partidaCriada: ${JSON.stringify(partidaCriada)}`)  
31     /*  
32      Recuperamos o ID da partida  
33     */  
34     const result = await partidaCriada.save()  
35     this.logger.log(`result: ${JSON.stringify(result)}`)  
36     const idPartida = result._id  
37     /*  
38      Com o ID do desafio que recebemos na requisição, recuperamos o  
39      desafio.  
40     */  
41     const desafio: Desafio = await this.clientDesafios  
42       .send('consultar-desafios',  
43         { idJogador: '', _id: partida.desafio })  
44       .toPromise()  
45     /*  
46      Acionamos o tópico 'atualizar-desafio-partida' que será  
47      responsável por atualizar o desafio.  
48     */  
49     return this.clientDesafios  
50       .emit('atualizar-desafio-partida',  
51         { idPartida: idPartida, desafio: desafio })  
52       .toPromise()  
53  
54   } catch (error) {  
55     this.logger.error(`error: ${JSON.stringify(error.message)}`)  
56     throw new RpcException(error.message)  
57   }  
58  
59 }
```


Observações:

- Lembre-se de envolver seus métodos com o bloco try/catch, lançando uma RpcException, em caso de erro;
- Como iremos criar um event emitter, será necessário criar nosso modulo proxyrmq neste projeto.

iii. **Migrando** a classe **PartidasModule**

Lembre-se de importar o MongooseModule, registrando o PartidaSchema

iv. Criando o método **criarPartida** da classe **PartidasController**

Utilize um event subscriber para recuperar as mensagens que estão no RabbitMQ

```
14
15     @EventPattern('criar-partida')
16     async criarPartida(
17         @Payload() partida: Partida,
18         @Ctx() context: RmqContext
19     ) {
20         const channel = context.getChannelRef()
21         const originalMsg = context.getMessage()
22         try {
23             this.logger.log(`partida: ${JSON.stringify(partida)}`)
24             await this.partidasService.criarPartida(partida)
25             await channel.ack(originalMsg)
26         } catch(error) {
27             this.logger.log(`error: ${JSON.stringify(error.message)}`)
28             const filterAckError = ackErrors.filter(
29                 ackError => error.message.includes(ackError))
30             if (filterAckError) {
31                 await channel.ack(originalMsg)
32             }
33         }
34     }
```

Observações:

- Não se esqueça de envolver seus métodos com o bloco try/catch/finally
- Não se esqueça de aplicar o acknowledge nas mensagens.