05

Construindo um codec

Transcrição

Precisaremos criar um *codec* para que o Mongo consiga lidar com nossos objetos Java, mas como fazemos isso? Inicialmente criaremos uma classe `AlunoCodec` no pacote `br.com.alura.escolalura.codecs`. Esta classe precisará implementar uma interface chamada `CollectibleCodec` e, em seu *generic*, informamos o tipo do *codec*. O código abaixo apresenta a classe `AlunoCodec` com os métodos da interface ainda não implementados:

```
package br.com.alura.escolalura.codecs;

public class AlunoCodec implements CollectibleCodec<Aluno>{

  @Override
  public void encode(BsonWriter arg0, Aluno arg1, EncoderContext arg2) {
  }

  @Override
  public Class<Aluno> getEncoderClass() {
    return null;
  }

  @Override
  public Aluno decode(BsonReader arg0, DecoderContext arg1) {
    return null;
  }

  @Override
  public boolean documentHasId(Aluno arg0) {
    return false;
  }

  @Override
  public Aluno generateIdIfAbsentFromDocument(Aluno arg0) {
    return null;
  }

  @Override
  public BsonValue getDocumentId(Aluno arg0) {
    return null;
  }

}
```

Veremos cada método um a um e iniciaremos a implementação do método `encode`, responsável por converter o objeto Java em um documento. Nele criaremos um documento e o popularemos com os valores dos atributos do objeto `aluno`. Observe:

```
public void encode(BsonWriter writer, Aluno aluno, EncoderContext encoder) {
  ObjectId id = aluno.getId();
```

```
    String nome = aluno.getNome();
    Date dataNascimento = aluno.getDataNascimento();
    Curso curso = aluno.getCurso();

    Document documento = new Document();
    documento.put("_id", id);
    documento.put("nome", nome);
    documento.put("data_nascimento", dataNascimento);
    documento.put("curso", new Document("nome", curso.getNome()));

  }
```

Para finalizarmos a codificação, precisaremos de um *codec* do tipo `Document`, sendo assim, pedimos para esse *codec* receber o documento que acabamos de criar, o que será feito diretamente no construtor da nossa classe. O atribuiremos a um atributo de classe utilizando seu método `encode` para realizarmos o restante do processo. Para este método precisaremos passar o objeto `writer`, o `documento` criado pelo método, e o `encoder`, que recebemos na assinatura do mesmo. Assim, teremos:

```
public class AlunoCodec implements CollectibleCodec<Aluno>{

  private Codec<Document> codec;

  public AlunoCodec(Codec<Document> codec) {
    this.codec = codec;
  }

  @Override
  public void encode(BsonWriter writer, Aluno aluno, EncoderContext encoder) {
    // código omitido

    codec.encode(writer, documento, encoder);
  }
  // código omitido
}
```

Com isto pronto, poderemos passar à implementação do próximo método, o `getEncoderClass`, no qual precisaremos retornar apenas a classe à qual o nosso *codec* fará a conversão - neste caso, a classe `Aluno`:

```
@Override
public Class<Aluno> getEncoderClass() {
  return Aluno.class;
}
```

O próximo método a ser implementado é o `decode`, ou seja, que trata do processo inverso do `encode`. Nele, precisaremos escrever o código que transforma um documento em um objeto `aluno`. Faremos isto posteriormente, deixaremos o método como ele está.

```
@Override
public Aluno decode(BsonReader arg0, DecoderContext arg1) {
  return null;
}
```

O método que segue verifica se o aluno possui *id*, e nele receberemos um objeto `aluno`, e retornaremos apenas uma expressão lógica que resulta em *true* ou *false*, comparando o atributo `id` com `null`.

```
@Override
public boolean documentHasId(Aluno aluno) {
  return aluno.getId() == null;
}
```

O método `generateIdIfAbsentFromDocument` é responsável por criar um *id* caso o documento não o tenha. Porém, ele não gera o *id* em si. O que faremos é verificar se o *id* existe com o método `documentHasId`. Caso não exista, criaremos o *id* com o método `criaId` da classe `Aluno` (*criaremos este método também*) e, caso o *id* exista, retornaremos o aluno.

```
@Override
public Aluno generateIdIfAbsentFromDocument(Aluno aluno) {
  return documentHasId(aluno) ? aluno.criaId() : aluno;
}
```

O método `criaId` só precisa usar o método `setId` passando um novo `ObjectId` como argumento retornando a si mesmo, afinal o método `generateIdIfAbsentFromDocument` espera retornar um aluno.

```
public Aluno criaId() {
  setId(new ObjectId());
  return this;
}
```

O último método é responsável por retornar um objeto `BsonValue` com o *id* do documento. Primeiro faremos uma verificação da existência do *id*, caso não, lançaremos uma exceção do tipo `IllegalStateException`, indicando o estado ilegal do objeto, e como mensagem indicaremos que o documento não possui *id*. Caso o *id* exista, converteremos o mesmo para uma base hexadecimal, entendida pelo Mongo, envolvendo-a em um objeto `BsonString`:

```
@Override
public BsonValue getDocumentId(Aluno aluno) {
  if(!documentHasId(aluno)){
    throw new IllegalStateException("Esse Document não tem id");
  }
  return new BsonString(aluno.getId().toHexString());
}
```

Agora podemos testar! Espere, podemos mesmo? Criamos o *codec*, isso é fato, mas não indicamos em lugar nenhum que o utilizaríamos. Isso precisa ser feito no momento da conexão, indicando que o Mongo terá este *codec* disponível.

Criaremos o *codec* de `Document`, necessário para ser utilizado por nossa classe - lembra-se que ela recebe um *codec* no construtor? A classe `MongoClient` tem um método chamado `getDefaultCodecRegistry` no qual poderemos capturar o *codec* de `Document`. No método `salvar` da classe `AlunoRepository` teremos:

```
Codec<Document> codec = MongoClient.getDefaultCodecRegistry().get(Document.class);
```

Agora sim, podemos criar uma instância do `AlunoCodec`:

```
Codec<Document> codec = MongoClient.getDefaultCodecRegistry().get(Document.class);
AlunoCodec alunoCodec = new AlunoCodec(codec);
```

Com o objeto `alunoCodec`, precisaremos incluí-lo em um registrador de *codecs* que precisa não apenas ter o nosso *codec* mas também os outros do próprio Mongo.

```
Codec<Document> codec = MongoClient.getDefaultCodecRegistry().get(Document.class);
    AlunoCodec alunoCodec = new AlunoCodec(codec);

    CodecRegistry registro = CodecRegistries.fromRegistries(
        MongoClient.getDefaultCodecRegistry(),
        CodecRegistries.fromCodecs(alunoCodec));
```

Ainda precisaremos de mais dois passos para termos o *codec* como opção do Mongo. Com o registro de todos eles no objeto `registro`, criaremos um objeto do tipo `MongoClientOptions` para que, no momento da conexão com o Mongo, como opção, ele tenha todos os *codecs* necessários, inclusive o nosso. A classe `MongoClientOptions` nos ajuda nessa tarefa.

```
MongoClientOptions options = MongoClientOptions.builder().codecRegistry(registro).build();
```

Note que a única utilidade foi do registro dos *codecs* nas opções do **builder*. Agora, a partir do objeto `options`, basta passá-lo como argumento no construtor da classe `MongoClient`, porém, como trata-se de um segundo parâmetro, somos obrigados a passar o primeiro, endereço e porta da conexão.

```
MongoClient cliente = new MongoClient("localhost:27017", options);
```

O método `salvar` da classe `AlunoRepository` após todos esses passos fica da seguinte forma:

```
public void salvar(Aluno aluno){
  Codec<Document> codec = MongoClient.getDefaultCodecRegistry().get(Document.class);
  AlunoCodec alunoCodec = new AlunoCodec(codec);

  CodecRegistry registro = CodecRegistries.fromRegistries(
      MongoClient.getDefaultCodecRegistry(),
      CodecRegistries.fromCodecs(alunoCodec));

  MongoClientOptions options = MongoClientOptions.builder().codecRegistry(registro).build();

  MongoClient cliente = new MongoClient("localhost:27017", options);
  MongoDatabase bancoDeDados = cliente.getDatabase("test");
  MongoCollection<Aluno> alunos = bancoDeDados.getCollection("alunos", Aluno.class);
  alunos.insertOne(aluno);


}
```

Se testarmos agora não teremos nenhum erro! Porém, será que está funcionando mesmo? Cadastrando a aluna Júlia do curso de Administração, e encontramos um erro. Consultando o Mongo manualmente, teremos:

```
{
  "_id" : ObjectId("5998946e161a4b2792ed78fc"),
  "nome" : "Julia",
  "data_nascimento" : ISODate("2017-08-13T03:00:00Z"),
  "curso" : {
    "nome" : "Administração"
  }
}
```

Ótimo! Tudo funciona como esperado! Vimos que criar um *codec* não é nada complicado, e sim trabalhoso, porém ajuda bastante no trabalho de conversão dos documentos Mongo para objetos Java.