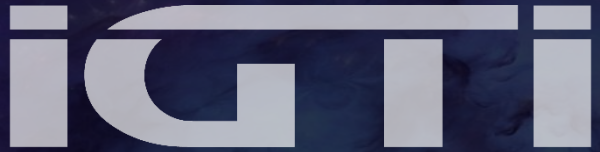


Persistência e Pesquisa de Dados

Capítulo 2. Mapeamento Objeto Relacional

Prof. Gustavo Aguilar

Aula 2.1. Introdução ao Mapeamento Objeto Relacional

Nesta aula

- Motivação do Mapeamento Objeto Relacional.
- O que é Mapeamento Objeto Relacional?
- Regras de Mapeamento Objeto Relacional.

Motivação do Mapeamento Objeto Relacional

- Conceitos de programação orientada a objetos (POO):
  - Alguns anos antes da publicação de Codd acerca do modelo relacional;
  - Linguagem Simula 67, por Ole Johan e Kristen Nygaard, em 1967;
  - Termo criado oficialmente em 1971 ➔ Alan Kay, linguagem *Smaltalk*.
- Viralização dos SGBDs relacionais:
  - Vertente para atender a demanda da programação orientada a objetos;
  - Década de 80 ➔ **SGBDOO.**

Motivação do Mapeamento Objeto Relacional

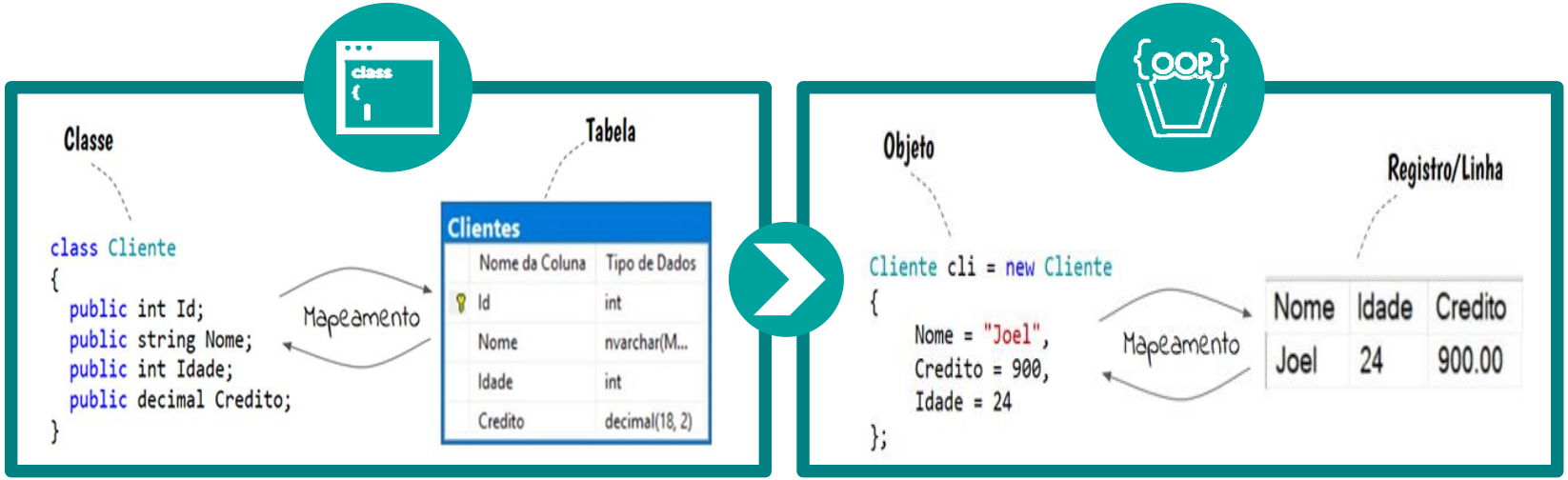
- Vantagens de um **banco de dados orientado a objetos** (**BDOO**)
  - Persistência transparente (mais performática);
  - Menos codificação e grande facilidade de abstração da realidade.
- Grande quantidade de artigos publicados pelo projeto *ORION;*
- Campanha comercial massiva em cima do *GemStone;*
- Entretanto, os sistemas gerenciadores de banco de dados orientado a objetos não chegaram perto da popularidade dos SGBDRs
  - Mais confiáveis e eficientes;
  - Regras e rotinas para persistência ➔ embutidas no próprio banco.

O Que É Mapeamento Objeto Relacional?

- Do inglês *Object Relational Mapping* ➔ **ORM**;
- Técnica usada para conversão de dados entre linguagens de programação orientadas a objetos e bancos de dados relacionais;

O que é Mapeamento Objeto Relacional?

- Na prática: associação entre classes e objetos com tabelas e linhas;
- Nível de código da aplicação ➔ trabalha-se com classes e objetos;
- Nível do banco de dados ➔ traduzidos para tabelas e linhas;



O que é Mapeamento Objeto Relacional?

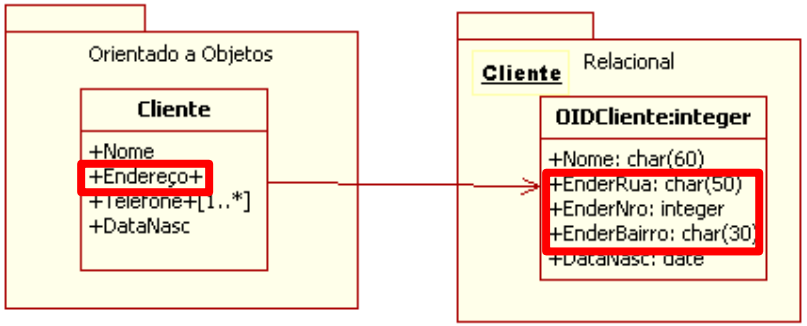
- Mapeamento ➔ **Camada de Persistência de Objetos;**
  - Uma biblioteca ou trecho de código;
  - Transparência do processo de persistência de dados;
  - Armazenamento em meio não volátil:
    - ORM ➔ em banco de dados relacional;
  - Para o programador:
    - Abstração dos comandos SQL do SGBD;
    - Como se estivesse em um ambiente 100% OO.

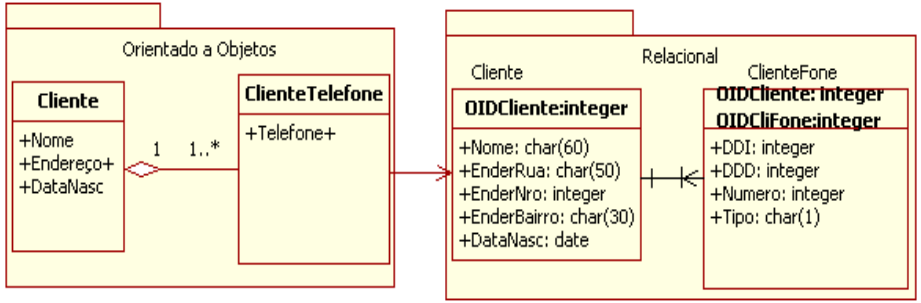
Regras de Mapeamento Objeto Relacional

- Criação de uma **tabela** para cada **classe;**
- Criação de uma **coluna** para cada **atributo simples** do objeto;
- Criação de uma **coluna** para o **OID** (*Object Identifier*);
- Definição de uma **chave primária** para cada tabela: **cada objeto é único** e garantido pelo OID.
  - OID como chave primária ➔ garante a unicidade dos registros na tabela.

Regras de Mapeamento Objeto Relacional

- **Atributos compostos:**
  - mapeados em **várias colunas,** ou
  - **consolidados em apenas uma.**
- **Atributos multivalorados:**
  - **Nova tabela** (entidade fraca)
  - **Chave primária**:
    - **Chave estrangeira** (PK da tabela criada para a classe com o atributo multivalorado)
    - + **PK da nova tabela**.

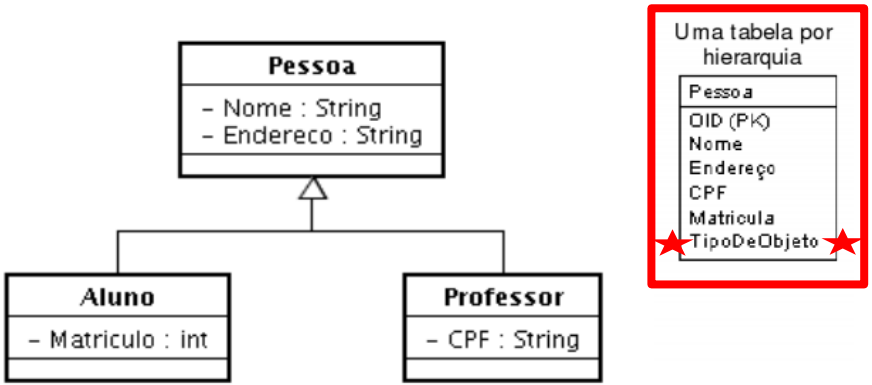




Regras de Mapeamento Objeto Relacional

- **Mapeamento de herança ➔** três abordagens.

1) **Criar uma tabela para cada hierarquia e uma coluna para cada atributo das classes dessa hierarquia. Criar uma coluna adicional para diferenciar a classe à qual o dado pertence.**



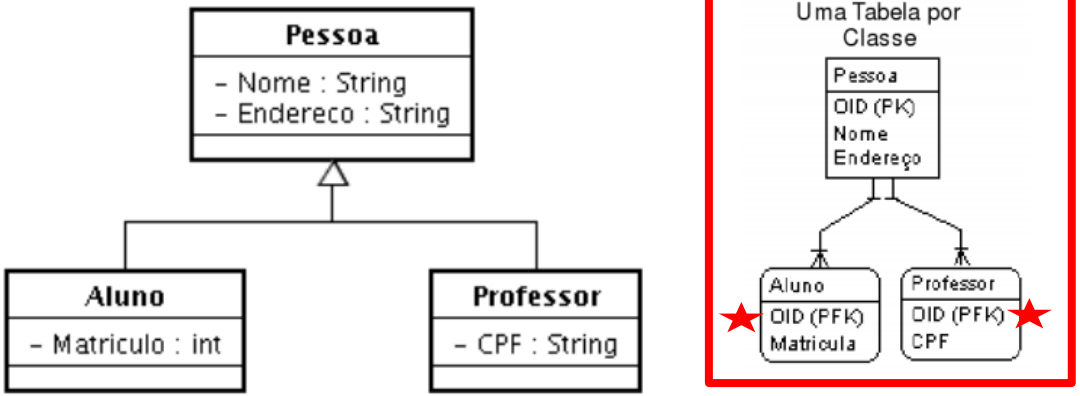
Regras de Mapeamento Objeto Relacional

- Mais **performático** na manipulação dos dados ➔ **todos os atributos** da hierarquia em uma **única tabela**.

- Ao persistir um objeto da hierarquia no banco de dados, será necessário **persistir também, vazios**, os atributos das demais classes ➔ **tupla** com **vários campos sem utilidade**.

Regras de Mapeamento Objeto Relacional

2) **Criar uma tabela para cada classe (especialização) e uma coluna em cada tabela especializada para ser a chave estrangeira do relacionamento com a tabela pai (classe abstrata).**



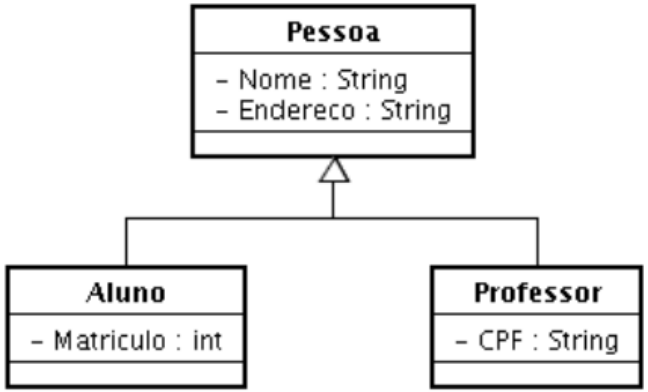
Regras de Mapeamento Objeto Relacional

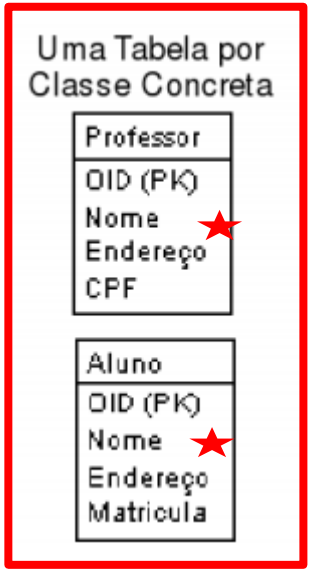
- Reflete mais **naturalmente** a **hierarquia** das classes;
- Mantém **normalização** que existir.

- Menos performática ➔ **mais tabelas** ➔ **mais *joins*** para recuperação dos dados;
- Mais **complexidade** nas queries
  - Minimizado com a criação de visões.

Regras de Mapeamento Objeto Relacional

3) **Criar uma tabela para cada classe concreta e, em cada tabela, criar uma coluna para cada atributo herdado da classe abstrata, além dos atributos específicos da classe concreta.**





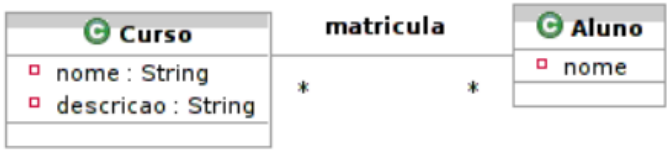
Regras de Mapeamento Objeto Relacional

- **Facilidade** de **manipulação** ➔ todos os dados de uma classe em apenas uma tabela;
- Menos joins ➔ mais **performance**.

- **Redundância** dos dados comuns ➔ quando um objeto existir em mais de uma classe concreta.
- **Trabalho extra** ➔ modificações na classe abstrata requerem que as tabelas geradas para as classes concretas também sejam alteradas.

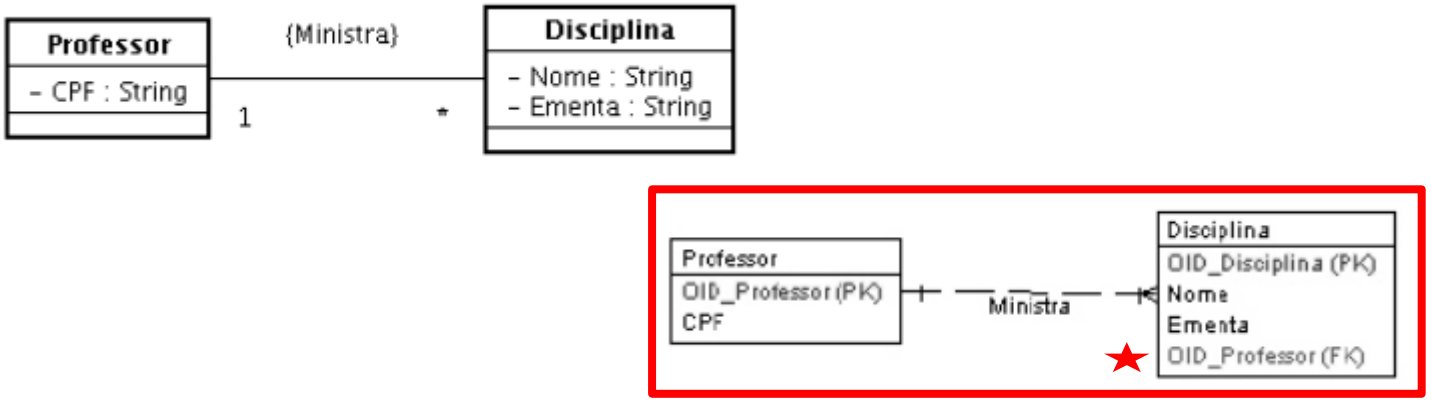
Regras de Mapeamento Objeto Relacional

- **Associação muitos-para-muitos** (relacionamento N:N)
  - Uma tabela para cada classe envolvida;
  - Criar **uma tabela associativa** para cada associação;
  - **Chave primária da tabela associativa ➔ Chaves estrangeiras**.



Regras de Mapeamento Objeto Relacional

- **Associação um-para-um** (relacionamento 1:1)
  - Criar **uma tabela para cada classe envolvida;**
  - Criar **uma coluna** para o atributo identificador da classe referenciada, na tabela que o referencia, para ser a chave estrangeira.



Regras de Mapeamento Objeto Relacional

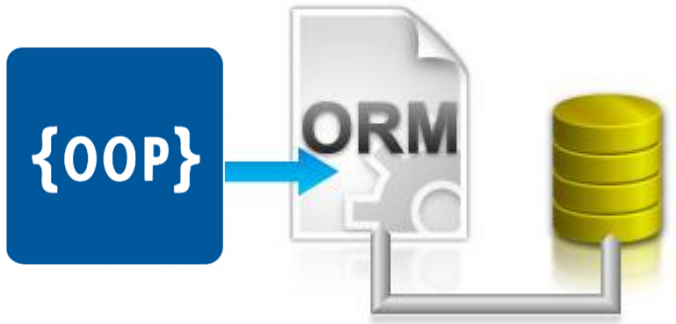
- **Associações um-para-muitos** (1:N)
  - Utilizar a mesma regra da associação 1:1
  - Criar **uma tabela para cada classe envolvida;**
  - Criar **uma coluna** para ser a chave estrangeira.
- **Associação todo-parte** ➔ agregação / composição ➔ mesmas estratégias para mapeamento de herança:
  - Criar uma tabela para cada hierarquia; ou
  - Criar uma tabela para cada classe; ou
  - Criar uma tabela para cada classe concreta.

O que é Mapeamento Objeto Relacional?

- Mapeamento concluído (nível de modelo de dados) ➔ definir como será feita a persistência dos dados:
  - De forma manual pelo programador ➔ haja código !!!

OU

  - **Com uma Ferramenta ORM:**
    - Mapeamento configurado na ferramenta;
    - Persistência feita implicitamente pelo framework;
    - Sem o ônus de construção de código para isso;
    - Mais segurança e confiança em todo o processo de persistência de dados.



Conclusão

- ☑ Aproveitamento do melhor dos dois mundos, SGBDRs e programação OO como motivadores do mapeamento objeto relacional.
- ☑ O Mapeamento Objeto Relacional é uma técnica usada para conversão de dados entre esses dois mundos.
- ☑ Regras de Mapeamento Objeto Relacional para orientar essa conversão.

Próxima aula

- Ferramentas de Mercado.

Aula 2.2. Ferramentas de Mercado

Nesta aula

- Ferramentas ORM de Mercado.

Ferramentas ORM de Mercado

- **Sem** uma **vastidão** de ferramentas ORM (como acontece com as ferramentas case para modelagem de dados);
- **Pluralidade suficiente** para atender as **linguagens** de programação mais **utilizadas**.

- **Hibernate** HIBERNATE
  - ✓ Gratuita;
  - ✓ Para linguagem ***Java***;
  - ✓ http://hibernate.org/orm/.
- **NHibernate** NHibernate The object-relational mapper for .NET
  - ✓ Gratuita;
  - ✓ Para ***.NET***;
  - ✓ https://nhibernate.info/.

Ferramentas ORM de Mercado

- **Django** 
  - ✓ Gratuita;
  - ✓ Para linguagem ***Python***;
  - ✓ www.djangoproject.com/.
- **Laravel** 
  - ✓ Gratuita;
  - ✓ Para linguagem ***PHP***;
  - ✓ https://laravel.com/.
- **LLBLGen Pro** | 
  - ✓ **Paga**;
  - ✓ Para ***.NET***;
  - ✓ http://www.llblgen.com/.
- **Ruby on Rails** 
  - ✓ Gratuita;
  - ✓ Para linguagem ***Rubi***;
  - ✓ https://rubyonrails.org/.
- **Entity Framework** 
  - ✓ Gratuita / Paga (extensões);
  - ✓ Para ***.NET***;
  - ✓ https://docs.microsoft.com/e.
- **Sequelize**
  - ✓ Gratuita;
  - ✓ Para ***Node.js***;
  - ✓ http://docs.sequelizejs.com/.

Conclusão

☑ Há uma quantidade razoável de boas ferramentas ORM de Mercado, a maioria delas gratuita.

Próxima aula

- Introdução ao Sequelize.

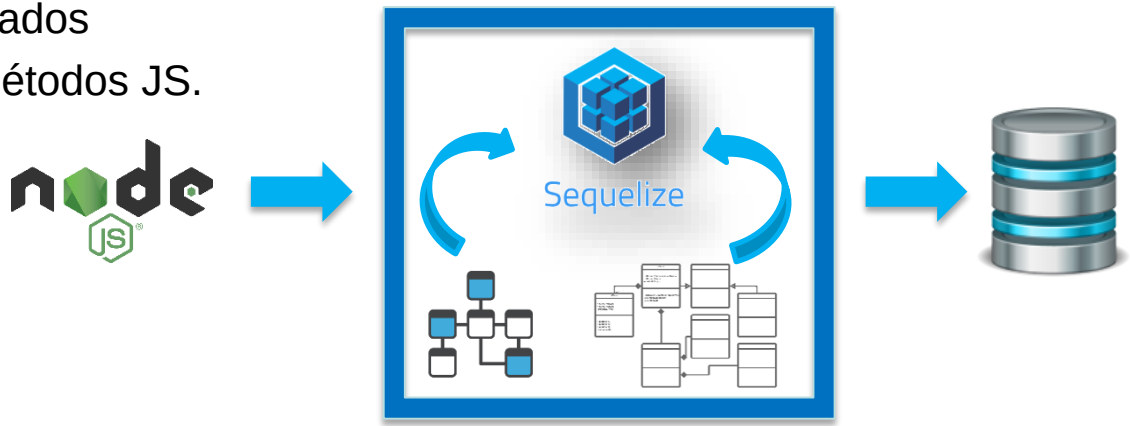
Aula 2.3. Introdução ao Sequelize

Nesta aula

- Introdução ao Sequelize.

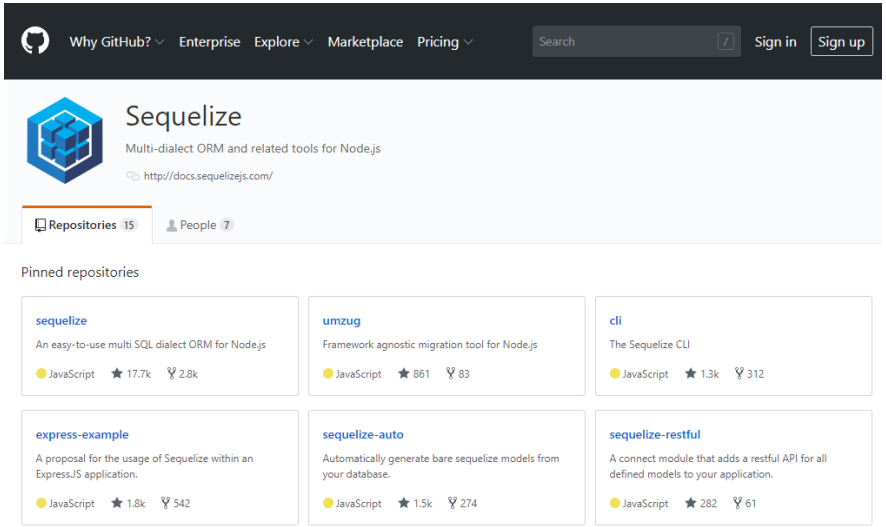
Introdução ao Sequelize

- Ferramenta ORM **open source p**ara **Node.js** *(*versão 4 / superior);
- Multidialeto (comandos SQL):
  - *MySQL / MariaDB, PostgreSQL, SQLite* e *SQL Server*.
- Objetos **JavaScript** mapeados em tabelas, colunas e linhas:
  - Manipulação de dados relacionais com métodos JS.



Introdução ao Sequelize

- Apesar de **gratuita** ➔ suporte a **transações**, **relacionamentos**, leitura em cenários de **replicação** e outros recursos avançados;
- Boa comunidade no **GitHub:**
  - https://github.com/sequelize.
- Ótima **documentação:**
  - http://docs.sequelizejs.com.



Conclusão

☑ O Sequelize, apesar de gratuito, é multidialeto, com suporte a recursos avançados de orientação a objeto, uma comunidade boa no GitHub e uma ótima documentação.

Próxima aula

- Instalação e configuração do Sequelize para NodeJS.

Aula 2.4. Instalação e Configuração do Sequelize para NodeJS

Nesta aula

- Instalação e configuração do Sequelize para NodeJS.
- Teste de conexão.

Instalação e Configuração do Sequelize

- Linux / Windows ➔ servidor físico / virtual / docker;
- SGBD ➔ pré-requisito;
- Pode ser instalado via NPM ou YARN;
- Cenário escolhido para os exemplos desse curso:
  - Servidor virtual no Microsoft Azure com Linux Ubuntu 18;
  - MySQL 5.7, Sequelize 4.43.0
  - NodeJS v8.10 + Sequelize-cli 5.4 + Express 4.16.4 + Body-Parser 1.18.3
  - Instalação do Sequelize via NPM
  - Modelo de dados físico: Sist. Controle de Equipamentos de Informática.

Instalação e Configuração do Sequelize

1. Atualizar os repositórios e pacotes de sistema:

   sudo apt update

   sudo apt upgrade

2. Inicializar o projeto

   i. Criar o diretório para o projeto ➔ ***mkdir equipamentos***

   ii. Entrar no diretório ➔ ***cd equipamentos***

   iii. Inicializar o projeto ➔ ***npm init –y***

   iv. Verificar arquivo JSON criado ➔ ***package.json***

```
gustavoaagl@ul8seq4:~/equipamentos$ cat package.json
{
  "name": "equipamentos",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
```

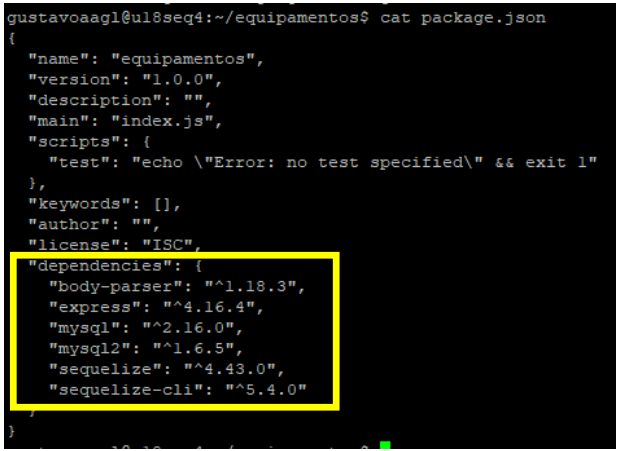
Instalação e Configuração do Sequelize

3. Instalar o ***NodeJS*** ➔ ***sudo apt-get install nodejs***
4. Instalar o ***Sequelize,*** a ***CLI,*** o ***Express*** e o ***Body-Parser:***

   npm install --save sequelize express body-parser sequelize-cli
5. Instalar o ***dialeto MySQL:***

   npm install --save mysql2
6. Verificar dependências:

   cat package.json



Teste de Conexão

- Método ***authenticate()*;**
- Parâmetros básicos de entrada:
  - ***database_name, username, password***
  - ***host***
  - ***dialect***
  - ***port***

Teste de Conexão

- Exemplo ➔ ***teste.js***

```
var Sequelize = require('sequelize')
  , sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
                        {
                                                host: 'localhost',
                                                dialect: "mysql",
                                                port:    3306,
                                                operatorsAliases: false
                                               } );

sequelize
  .authenticate()
  .then(function(err)  { console.log('SUCESSO!!! Conexao estabelecida.');},
                                function (err)
                                { console.log('ERRO!!! Nao foi possivel conectar.', err); });
```

- Executar ➔ ***nodejs*** nome_do_arquivo.js

```
gustavoaagl@u18seq4: ~/equipamentos
gustavoaagl@ul8seq4:~/equipamentos$ vi teste.js
gustavoaagl@ul8seq4:~/equipamentos$ nodejs teste.js
Executing (default): SELECT 1+1 AS result
SUCESSO!!! Conexao estabelecida.
```

Conclusão

☑ Sequelize roda no Linux e no Windows.

☑ Instalação do Sequelize é bem simples (NPM / YARN), mas requer que alguns pré-requisitos sejam instalados antes (SGBD por exemplo).

☑ Método ***authenticate*** para conectar no banco.

Próxima aula

- Dialetos e opções da Engine.

Aula 2.5. Dialetos e opções da Engine

Nesta aula

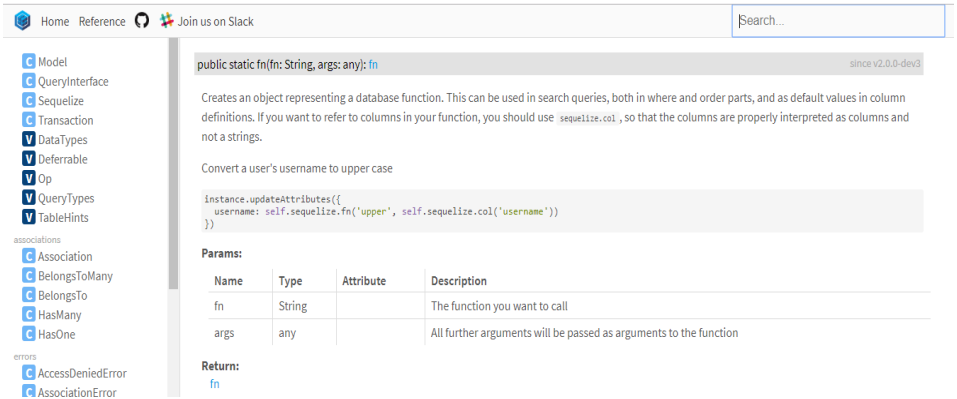
- API Sequelize.
- Dialetos.
- Opções da Engine.
- Suporte para Replicação.
- Execução de Query SQL (RAW).

API Sequelize

- Codificada a partir da classe principal ***sequelize***;
- Para usá-la, basta importa-la no código Java:

  const Sequelize = require ('sequelize');

- Inúmeros **métodos** para:
  - Definir modelos;
  - Mapear objetos;
  - Persistir e manipular dados.
- Documentação completa:
  - http://docs.sequelizejs.com/class/lib/sequelize.js~Sequelize.html.



Dialetos

- **Dialeto** ➔ SGBD que será utilizado;
- Biblioteca de conexão para o dialeto deve ser instalada;
  - Não é necessário importa-la ➔ o Sequelize se incube disso automaticamente.
  - *npm install --save pg pg-hstore* ➔ para PostgreSQL
  - *npm install --save mysql2* ➔ para MySQL
  - *npm install --save sqlite3* ➔ para SQLite
  - *npm install --save mariasql* ➔ para MariaDB
  - *npm install --save tedious* ➔ para SQL Server

Opções da Engine

- As principais opções da classe ***Sequelize:***
  - Parâmetros com o n**ome do banco, nome do usuário e senha**;
  - Requisitadas pelo método ***authenticate*** para abrir a conexão com o banco:
    - *dialect* ➔ sistema gerenciador de banco de dados que será usado*;*
    - *host* ➔ hostname ou ip onde está o banco de dados;
    - *port* ➔ porta na qual o banco de dados está respondendo;
    - *operatorsAliases* ➔ setado para *false,* evita erros de parse.

Opções da Engine

- Especificar opções do dialeto ➔ p. ex. conjunto de caracteres;

```
  dialectOptions:
{    socketPath: '/Applications/MAMP/tmp/mysql/mysql.sock',
     supportBigNumbers: true,
     bigNumberStrings: true,
     charset: 'utf8',
     dialectOptions: { collate: 'utf8_general_ci' }
}
```

- Configuração do pool de conexões do banco de dados (default é 1);

```
pool: {     max: 25,
            idle: 30000,
            acquire: 60000,
        }
```

Suporte para Replicação

- Opção disponível para trabalhar com **bancos de dados distribuídos:**
  - **Homogêneos** ➔ mesmo SGBD.
- **Configurações gerais aplicáveis a todas as réplicas:**
  - Não é necessário fornecê-las para cada instância.
- Sequelize usa **pool para gerenciar conexões com as réplicas:**
  - **Dois** pools internos, criados usando-se a configuração default do pool ou não;
  - Operação de escrita ou instrução específica para usar a réplica primária (*useMaster: true*) ➔ query utilizará o pool de gravação;
  - Para SELECT ➔ pool de leitura ➔ leituras distribuídas usando *round robin*.

Suporte para Replicação

```
var sequelize = new Sequelize('database', null, null,
{ dialect: 'mysql',
  port: 3306
  replication:
{   read:
    [  { host: '8.8.8.8', username: 'read-username', password: 'some-password' },
       { host: '9.9.9.9', username: 'another-username', password: null }
    ],
     write: { host: '1.1.1.1', username: 'write-username', password: 'any-password' }
},
  pool: {  max: 20,
           idle: 30000
        }
} )
```

Execução de Query SQL (RAW)

- Flexibilidade para execução de query SQL nativa do SGBD de dentro do código Java;
- Chamadas de ***raw SQL queries;***
- Executadas através do método ***query()*.**

```
sequelize.query("SELECT * FROM TipoEquipamento")
        .then(RegistrosTabela => {console.log(RegistrosTabela) } )
```

Execução de Query SQL (RAW)

- Possível também mapear facilmente uma query para um modelo predefinido ➔ opção ***Model.***

```
sequelize.query('SELECT * FROM Equipamento', { model: Equipamentos })
  .then(equipamentos => { console.log(equipamentos) })
```

- Após a execução do comando acima ➔ cada registro da **tabela *Equipamento*** se encontrará mapeado para o **modelo *Equipamentos.***

Conclusão

- ☑ API construída a partir da classe principal sequelize.
- ☑ Dialetos precisam ser instalados, mas não precisam ser instanciados manualmente.
- ☑ Opções da Engine úteis, como pool de conexão e collation.
- ☑ Suporte para bancos de dados distribuídos homogêneos e balanceamento de carga de leitura.
- ☑ Execução de Query SQL (RAW) nativamente no SGBD.

Próxima aula

- Criando e usando um modelo de dados básico no Sequelize.

Aula 2.6. Criando e usando um Modelo de Dados Básico no Sequelize

Nesta aula

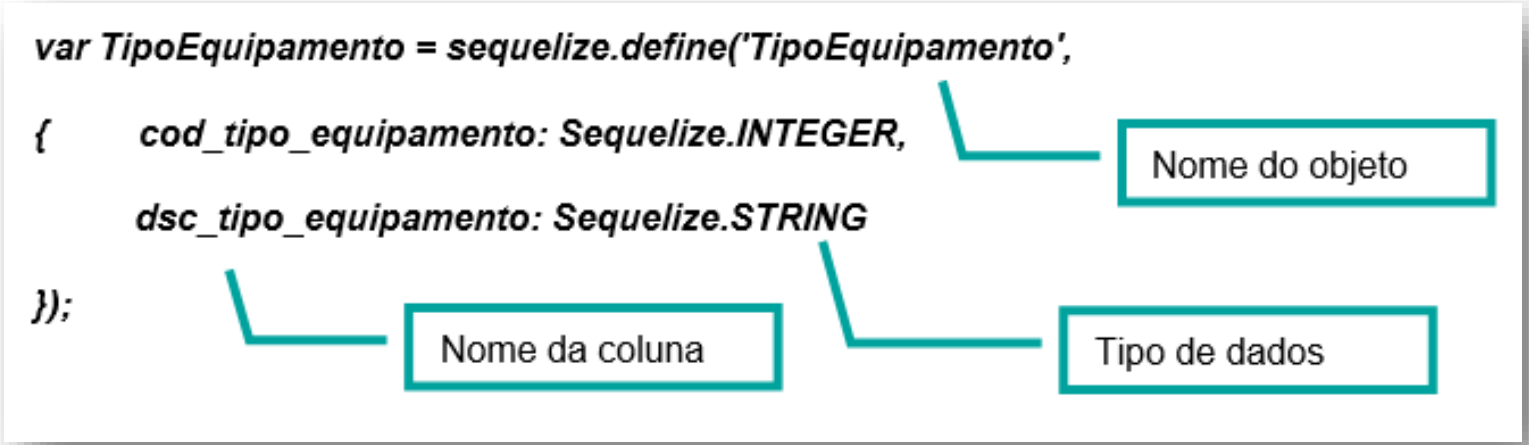
- Modos de Gerenciamento do Schema Físico.
- Definição do Modelo.
- Modelo Paranoid.
- Sincronização do Schema Físico.
- Criação e Persistência de Instâncias.
- Leitura de Dados.

Modos de Gerenciamento do Schema Físico

- **Com migrações** (***migrations***)
  - Similar ao controle de alterações em código (*GitHub/VisualStudio/SourceSafe);*
  - Transpõe o banco de dados de um estado (status) para outro e vice-versa;
  - Salvas em arquivos de migração ➔ possibilita reversões;
  - Mais tempo para configurar, porém fornece mais segurança e estabilidade:
    - Recomentada para ambientes com muitos servidores e/ou equipes grandes.
- **Automatizada**
  - O Sequelize cria as tabelas, de forma atômica, sem opção de rollback;
  - Boa opção para ambientes *stand alone* e/ou com equipe pequena e/ou testes;
  - Requer a especificação dos tipos de dados que se deseja persistir.

Definição do Modelo

- Feita usando o método ***define()***;
- O Sequelize tem suporte a vários tipos de dados:
  - http://docs.sequelizejs.com/manual/tutorial/models-definition.html#data-types



Definição do Modelo

- Por default, o Sequelize cria mais três colunas para controle interno:
  - ***Id int(11)*** ➔ **chave primária** da tabela (OID), **autoincremental**;
  - ***CreatedAt (datetime)*** ➔ preenchida quando uma instância do objeto é persistida na tabela;
  - ***updatedAt (datetime)*** ➔ preenchida quando uma coluna é atualizada.
- Pode-se optar por não gerar essas colunas de timestamp:

```
var TipoEquipamento = sequelize.define('TipoEquipamento',
        {
              cod_tipo_equipamento: Sequelize.INTEGER,
              dsc_tipo_equipamento: Sequelize.STRING
        },
        { timestamps: false})
```

Definição do Modelo

- Uma opção possível, é customizar o nome das colunas de timestamp;

```
var TipoEquipamento = sequelize.define('TipoEquipamento',
      {
            cod_tipo_equipamento: Sequelize.INTEGER,
            dsc_tipo_equipamento: Sequelize.STRING
      },
      {
            createdAt: 'dat_criacao',
            updatedAt: 'dat_ult_atualizacao'
      })
```

- Por default, o Sequelize gera o nome da tabela colocando um “s” ao final do nome do objeto;
  - Pode-se explicitar o nome da tabela;

```
var TipoEquipamento = sequelize.define('TipoEquipamento',
      {
            cod_tipo_equipamento: Sequelize.INTEGER,
            dsc_tipo_equipamento: Sequelize.STRING
      },
      { tableName: 'TipoEquipamento' })
```

Modelo Paranoid

- Opção útil para manter histórico dos registros;
- Registro não é deletado **fisicamente** da tabela quando o método de ***destroy*** (delete) é executado ➔ deleção lógica de dados;
- Utiliza o campo ***deletedAt***
  - Também pode ser renomeado;
  - Indica a data que o registro foi excluído.

```
var TipoEquipamento = sequelize.define('TipoEquipamento',
    {
        cod_tipo_equipamento: Sequelize.INTEGER,
        dsc_tipo_equipamento: Sequelize.STRING
    },
    {
        tableName: 'TipoEquipamento',
        paranoid: true,
        createdAt: 'dat_criacao',
        updatedAt: 'dat_ult_atualizacao',
        deletedAt: 'dat_exclusao'
    })
```

Sincronização do Schema Físico

- Etapa de criação da representação do modelo no banco de dados
  - Criação do **schema físico;**
  - Para os dados poderem ser persistidos;
  - Feito com o **método *sync ()***

```
sequelize.sync({ force: true })
  .then(function(err) { console.log('Tabela criado com sucesso!'); },
        function (err) { console.log('Erro ao criar a tabela.', err); });
```

Se a tabela existir ➔ dropa e a recria

Sincronização do Schema Físico

- Exemplo:

```
var Sequelize = require('sequelize')
  , sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
                    {                    host: 'localhost',
                                         dialect: "mysql",
                                         port:    3306,
                                         operatorsAliases: false
                    } );
sequelize.authenticate()
  .then ( function(err)  { console.log('SUCESSO!!! Conexao estabelecida.');},
            function (err) { console.log('ERRO!!! Nao foi possivel conectar.', err); }
          );

var TipoEquipamento = sequelize.define('TipoEquipamento',
{
        cod_tipo_equipamento: Sequelize.INTEGER,
        dsc_tipo_equipamento: Sequelize.STRING
});


sequelize.sync({ force: true })
  .then( function(err) { console.log('Tabela criada com sucesso!');},
           function (err){ console.log('Erro ao criar a tabela.', err); }
          );
```

Criação e Persistência de Instâncias

- Etapa de inserção dos dados ➔ persistência da instância de um objeto;
- Pode ser feita de duas maneiras:
  - **Em dois passos**:
    - Cria o objeto e depois o salva (métodos ***build*** e ***save***);
    - Mais flexível, com opções antes de persistir o objeto.

```
var tpequipamento = TipoEquipamento.build(
                    {
                      cod_tipo_equipamento: 01,
                      dsc_tipo_equipamento: 'Notebook'
                    } );

tpequipamento.save().then( function(err) { console.log('Registro inserido com sucesso!'); },
                           function (err){ console.log('Erro ao inserir o registro', err);}
                         );
```

Criação e Persistência de Instâncias

– **Com um passo**:

- Cria o objeto e já persiste (método ***create***);
- Mais performática.

```
var tpequipamento = TipoEquipamento.create(
                    {
                      cod_tipo_equipamento: 01,
                      dsc_tipo_equipamento: 'Notebook'
                    } )
                    .then( function(err) { console.log('Registro inserido com sucesso!'); },
                          function (err){ console.log('Erro ao inserir o registro', err);}
                     );
```

Criação e Persistência de Instâncias

- **Exemplo de persistência com um passo:**

```
var Sequelize = require('sequelize')
  , sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
                    {                   host: 'localhost',
                                        dialect: "mysql",
                                        port:    3306,
                                        operatorsAliases: false
                    } );

sequelize.authenticate()
  .then( function(err)  { console.log('SUCESSO!!! Conexao estabelecida.');},
         function (err) { console.log('ERRO!!! Nao foi possivel conectar.', err); } );

var TipoEquipamento = sequelize.define('TipoEquipamento',
              {  cod_tipo_equipamento: Sequelize.INTEGER,
                 dsc_tipo_equipamento: Sequelize.STRING
              },
              {      tableName: 'TipoEquipamento',
                     paranoid: true,
                     createdAt: 'dat_criacao',
                     updatedAt: 'dat_ult_atualizacao',
                     deletedAt: 'dat_exclusao'
              });
var tpequipamento = TipoEquipamento.create(
                     {  cod_tipo_equipamento: 01,
                        dsc_tipo_equipamento: 'Notebook'
                     } )
                     .then( function(err) { console.log('Registro inserido com sucesso!'); },
                           function (err){ console.log('Erro ao inserir o registro', err);} );
```

Leitura de Dados

- Feita usando o método ***findAll*** ➔ SELECT *.

```
var tpequipamento = TipoEquipamento.findAll()

        then (tpequipamento => {

                console.log(tpequipamento)

        } )
```

```
gustavoaagl@u18seq4: ~/equipamentos
gustavoaagl@u18seq4:~/equipamentos$ nodejs tipoequipamento_findAll.js
Executing (default): SELECT 1+1 AS result
Executing (default): SELECT `id`, `cod_tipo_equipamento`, `dsc_tipo_equipamento`, `dat_criacao`, `dat_ult_atualizacao`, `dat_exclus
ao` FROM `TipoEquipamento` AS `TipoEquipamento` WHERE (`TipoEquipamento`.`dat_exclusao` > '2019-03-09 20:16:42' OR `TipoEquipamento
`.`dat_exclusao` IS NULL);
SUCESSO!!! Conexao estabelecida.
[ TipoEquipamento {
    dataValues:
     { id: 1,
       cod_tipo_equipamento: 1,
       dsc_tipo_equipamento: 'Notebook',
       dat_criacao: 2019-03-09T20:14:43.000Z,
       dat_ult_atualizacao: 2019-03-09T20:14:43.000Z,
       dat_exclusao: null },
    _previousDataValues:
     { id: 1,
       cod_tipo_equipamento: 1,
       dsc_tipo_equipamento: 'Notebook',
```

Leitura de Dados

- Exemplo:

```
var Sequelize = require('sequelize')
  , sequelize = new Sequelize('BDEquipamentos', 'usrsequelize', 'sequelize',
                    {                   host: 'localhost',
                                        dialect: "mysql",
                                        port:    3306,
                                        operatorsAliases: false   } );
sequelize.authenticate()
  .then( function(err)  { console.log('SUCESSO!!! Conexao estabelecida.');},
         function (err) { console.log('ERRO!!! Nao foi possivel conectar.', err); } );

var TipoEquipamento = sequelize.define('TipoEquipamento',
              {  cod_tipo_equipamento: Sequelize.INTEGER,
                 dsc_tipo_equipamento: Sequelize.STRING
              },
              {       tableName: 'TipoEquipamento',
                      paranoid: true,
                      createdAt: 'dat_criacao',
                      updatedAt: 'dat_ult_atualizacao',
                      deletedAt: 'dat_exclusao'
              });


var tpequipamento = TipoEquipamento.findAll().

                        then (tpequipamento => {

                                                    console.log(tpequipamento)

                                                } )
```

Conclusão

- ☑ Duas formas existentes para gerenciamento do schema físico.
- ☑ A definição do modelo precisa ser feita para a forma automatizada.
- ☑ Modelo Paranoid para armazenamento de histórico físico dos dados.
- ☑ Sincronização do schema físico para criar as tabelas no banco.
- ☑ Criação e persistência de instâncias de objetos para inserir dados nas tabelas.
- ☑ Exemplo de leitura de dados com o método findAll().

Próxima aula

- Definição do Modelo de Dados.

Aula 2.7. Definição do Modelo de Dados

Nesta aula

- Opções para definição do Modelo de Dados.
- Validação do Modelo.
- Exportação e importação de Modelo.
- Índices.

Opções para Definição do Modelo de Dados

- Mapeamento entre objetos modelo OO e as tabelas em um SGBDR;
- Feita através do método ***define()***;

```
const Equipamento = sequelize.define('equipamento',
                                     {      cod_equipamento: Sequelize.INTEGER,
                                            dsc_equipamento: Sequelize.TEXT
                                     } )
```

- Definição de campo que não aceita ausência de valores (**NOT NULL**);

```
const Equipamento = sequelize.define('equipamento',
                          {      cod_equipamento: Sequelize.INTEGER,
                                 dsc_equipamento: {
                                                    type:Sequelize.TEXT,
                                                    allowNull: false
                                                  }
                          })
```

Opções para Definição do Modelo de Dados

- Atribuição de **valor default ➔** fixo ou dinâmico;

```
const Equipamento = sequelize.define('equipamento',
                    {       cod_equipamento: Sequelize.INTEGER,
                            dsc_equipamento: {
                                                type:Sequelize.TEXT,
                                                allowNull: false
                                              },
                            ind_ativo:      {   type: Sequelize.BOOLEAN,
                                                allowNull: false,
                                                defaultValue: true
                                              },
                            dat_compra:     {   type:Sequelize.DATE,
                                                defaultValue: Sequelize.NOW
                                            }
                    })
```

- Criação de **constraint única;**
  - Índice único.

```
nom_equipamento: {   type:Sequelize.STRING(500),
                     allowNull: false,
                     unique: true      }
```

Opções para Definição do Modelo de Dados

- Criação de **chave primária**;
  - o Sequelize não criará o campo **id** na tabela**.**

```
cod_equipamento: {    type:Sequelize.INTEGER,
                      allowNull: false,
                      primaryKey: true      }
```

- Definição de campo **autoincremental**;

```
cod_equipamento: {    type:Sequelize.INTEGER,
                      allowNull: false,
                      primaryKey: true,
                      autoIncrement: true   }
```

Opções para Definição do Modelo de Dados

- Criação de **chave estrangeira**:

```
const Equipamento = sequelize.define('equipamento',
                    {       cod_equipamento: Sequelize.INTEGER,
                            dsc_equipamento: {
                                    type:Sequelize.TEXT,
                                    allowNull: false
                                },
                            cod_tipo_equipamento:
                            {
                                type: Sequelize.INTEGER,
                                references:
                                {       //Nome do modelo "pai"
                                        model: TipoEquipamento,
                                        //Campo chave da tabela pai
                                        Key: 'id'
                                }
                            }
```

Validação do Modelo

- Visa fornecer **qualidade** e **consistência** dos dados;
- Validações de formato, conteúdo ou herança para cada atributo;
- Executadas automaticamente ➔ métodos *create, update* ou *save;*
- Validar manualmente ➔ chamar o método ***validate ()***;
- Sequelize fornece várias validações default:
  - Implementadas no arquivo ***validator.js.***
- http://docs.sequelizejs.com/manual/tutorial/models-definition.html#validations

Validação do Modelo

- Validar se a quantidade de itens da ordem de compra é no mínimo 1 dúzia e no máximo 10 dúzias.

```
OrdemCompra = Sequelize.define('ordemcompra',
{
        cod_ordem_compra: { type: Sequelize.INTEGER },
        qtd_itens:
        {                   type: Sequelize.INTEGER,
                            validate:
                            {
                                    notNull: true,
                                    min: 12,
                                    max: 120
                            }
        }
```

Validação do Modelo

- Também é possível definir uma validação customizada:

```
var OrdemCompra = sequelize.define('OrdemCompra',
{
        cod_ordem_compra: { type: Sequelize.INTEGER },
        qtd_itens:
        {
                type: Sequelize.INTEGER,
                validate:
                {
                        notNull:true,
                        min: 12,
                        max: 120,
                        ValorPar (value)
                        {
                            if (parseInt(value) % 2 != 0)
                            {
                                throw new Error('Apenas quantidades pares sao Aceitas!')
                            }
                        }
                }
        }
});
```

Exportação e Importação de Modelo

- Acelera o desenvolvimento ➔ não precisa definir o modelo toda vez;
- Evita erros na importação manual de um modelo;
- Exportar a definição de um modelo para um arquivo ➔ ***exports();***
- Importar a definição de um modelo ➔ ***import();***
- Performático:
  - A partir do Sequelize 1.5.0 o *import* é feito em cache.

```
module.exports = (sequelize, DataTypes) =>
{
        const TipoEquipamento =  sequelize.define('TipoEquipamento',
                        {
                                cod_tipo_equipamento: Sequelize.INTEGER,
                                dsc_tipo_equipamento: Sequelize.STRING },
                                {       tableName: 'TipoEquipamento',
                                        paranoid: true,
                                        createdAt: 'dat_criacao',
                                        updatedAt: 'dat_ult_atualizacao',
                                        deletedAt: 'dat_exclusao'
                                });
        return TipoEquipamento;
};
```

Exportação e Importação de Modelo

```
const TipoEquipamento =
sequelize.import ("/home/gustavoaagl/equipamentos/tipoequipamento_export.js")
```

Caminho e nome do arquivo com o modelo a ser importado

```
var Equipamento = sequelize.define('Equipamento',
{
        cod_equipamento: {   type:Sequelize.INTEGER,
                             allowNull: false,
                             autoIncrement: true,
                             primaryKey: true
                         },
        nom_equipamento: {          type:Sequelize.STRING(500),
                                    allowNull: false,
                                    unique: true
                         },
        cod_tipo_equipamento: {     type: Sequelize.INTEGER,
                                    references:
                                    {       model: TipoEquipamento,
                                            Key: 'id'
                                    }
                              }
```

Índices

- Definição dos índices no momento do mapeamento:

```
sequelize.define('user', {}, {
  indexes: [
    // Create a unique index on email
    {
      unique: true,
      fields: ['email']
    },

    // Creates a gin index on data with the jsonb_path_ops operator
    {
      fields: ['data'],
      using: 'gin',
      operator: 'jsonb_path_ops'
    },

    // By default index name will be [table]_[fields]
    // Creates a multi column partial index
    {
      name: 'public_by_author',
      fields: ['author', 'status'],
      where: {
        status: 'public'
      }
    },

    // A BTREE index with a ordered field
    {
      name: 'title_index',
      method: 'BTREE',
      fields: ['author', {attribute: 'title', collate: 'en_US', order: 'DESC', length: 5}]
    }
  ]
})
```

Conclusão

☑ Várias opções para definição do modelo de dados.

☑ Método de validação do modelo visando consistência e qualidade dos dados.

☑ Exportação e importação de modelo para agilizar e evitar erros na referenciação de modelos.

☑ Possibilidade de criar índices no momento do mapeamento, visando um schema físico mais performático.

Próxima aula

- Usando e consultando o Modelo de Dados.

Aula 2.8. Usando e consultando o Modelo de Dados

Nesta aula

- Consultas com filtro.
- Ordenação e agrupamento.
- Operadores.
- Funções de agregação.
- Deleção e atualização de Dados.

Consultas com Filtro

- Métodos correspondentes ao comando SELECT da linguagem SQL;
- Consultar por registros com **ID(s) específico(s):**

  *TipoEquipamento.**findByID (123)**.then (tipoequipamento => { })*

- Consultar **filtrando por atributos:**

  *Equipamento.**findOne**({ where: {nom_equipamento: 'Notebook'} })*

  .then(equipamento => { })

- Consultar **filtrando por atributos e os atributos retornados:**

  Equipamento.findOne({where: {nom_equipamento: 'Notebook'},attributes:

  ['cod_equipamento', 'nom_equipamento', 'ind_ativo']]}).then(equipamento => { })

Operadores

- Possibilitam a execução de queries mais complexas com operações combinatórias;

SELECT * FROM TipoEquipamento WHERE cod_tipo_equipamento IN (12, 13):

TipoEquipamento.findAll({ where: {cod_tipo_equipamento:

*{ **[Op.or]: [12, 13]** }}});*

```
[Op.and]: {a: 5}           // AND (a = 5)
[Op.or]: [{a: 5}, {a: 6}]  // (a = 5 OR a = 6)
[Op.gt]: 6,                // > 6
[Op.gte]: 6,               // >= 6
[Op.lt]: 10,               // < 10
[Op.lte]: 10,              // <= 10
[Op.ne]: 20,               // != 20
[Op.eq]: 3,                // = 3
[Op.not]: true,            // IS NOT TRUE
[Op.between]: [6, 10],     // BETWEEN 6 AND 10
[Op.notBetween]: [11, 15]  // NOT BETWEEN 11 AND 15
```

Funções de Agregação

- **Contar as Ocorrências de um Elemento ➔ COUNT:**

  Equivalente a SELECT COUNT(*) FROM Equipamento:

  *Equipamento.**count()**.then(**x** => {*

  *console.log("Existe um Total de " + **x** + " equipamentos!")})*

  Equivalente a SELECT COUNT (*) FROM Equipamento

  WHERE cod_equipamento > 10:

  *Equipamento.**count**({ **where**: {'cod_equipamento': **{[Op.gt]: 10}}** }).then(**x** => {*

  *console.log("Total de " + **x** + " equipamentos com código maior que 10.")})*

Funções de Agregação

- **Somar Valores de Atributos ➔ SUM();**

  Equivalente a SELECT SUM (vlr_equipamento) FROM Equipamento:

  *Equipamento.**sum**('vlr_equipamento').then(**sum** => { })*

- **Retornar o Valor Máximo de um Atributo ➔ MAX()**

  Equivalente a SELECT MAX (vlr_equipamento) FROM Equipamento

  WHERE ind_ativo IN (1,2):

  *Equipamento.**max**('vlr_equipamento',{where:{ind_ativo:*

  *{**[Op.or]:[1,2]**}}}).then(**max**=>{})*

Funções de Agregação

- **Retornar o Valor Mínimo de um Atributo ➔ MIN()**

  Equivalente a SELECT MIN (vlr_equipamento) FROM Equipamento:

  *Equipamento.**min**('vlr_equipamento').then(**min**=>{})*

Deleção e Atualização de Dados

- **Deletar** registros ➔ método ***destroy()***

  Equivalente a DELETE FROM Equipamento WHERE ind_ativo = 0:

  *Equipamento.**destroy**({ where: {ind_ativo: 0 } });*

- **Atualizar** registros ➔ método ***update()***

  Equivalente a UPDATE Equipamento SET updatedAt = null

  WHERE deletedAt NOT NULL:

  *Equipamento.**update**({ **updatedAt**: null,}, {where: {deletedAt: {[Op.ne]: null }}});*

Conclusão

☑ Sequelize fornece vários métodos e opções para as operações CRUD (**criar**, **recuperar**, **atualizar e deletar**) de um sistema gerenciador de banco de dados relacional.

Próxima aula

- Overview de associações e transações.

Aula 2.9. Overview de Associações e Transações

Nesta aula

- Associações.
- Transações.

Associações

- Recurso equivalente no Sequelize para as operações de relacionamento (***join***) no modelo relacional;
- Suporte a todas as cardinalidades existentes;
- **Relacionamento Um-Para-Um (1:1)**
  - Método ***hasOne()*** e ***belongsTo()***
    - *hasOne* insere a chave de associação no destino
    - *belongsTo* insere a chave de associação no modelo de origem.

```
const Equipamento = this.sequelize.define('equipamento', {/* atributos */})
const Serial  = this.sequelize.define('serial', {/* atributos */});

Equipamento.hasOne(Serial);

Serial.belongsTo(Equipamento);
```

Associações

- **Relacionamento Um-Para-Muitos (1:N)**
  - Métodos ***hasMany()*** e ***belongsTo()***;
  - *hasMany* insere a chave de associação no modelo de destino;
  - *belongsTo* insere a chave de associação no modelo de origem.

```
const Equipamento = this.sequelize.define('equipamento', {/* atributos */})
const TipoEquipamento  = this.sequelize.define('tipoequipamento', {/* atributos */});

TipoEquipamento.hasMany(Equipamento);

Equipamento.belongsTo(TipoEquipamento);
```

Associações

- **Relacionamento Muitos-Para-Muitos (N:N)**
  - Implementado pelo método ***belongsToMany()***.

```
const Equipamento = this.sequelize.define('equipamento', {/* atributos */})
const OrdemReparo  = this.sequelize.define('ordemreparo', {/* atributos */});

Equipamento.belongsToMany(OrdemReparo);

OrdemReparo.belongsToMany(Equipamento);
```

Transações

- Suporte ao controle transacional dos SGBDs relacionais:
  - Garantir ACID (atomicidade, consistência, isolamento e durabilidade);
  - Sequelize disponibiliza **duas maneiras para realizar transações.**

- **Transação Gerenciada (*auto-callback)***:
  - Confirmará ou reverterá automaticamente a transação com base no resultado de uma cadeia de queries;
  - O programador não precisa explicar o commit / rollback no código.

Transações

```
return sequelize.transaction(t => {
  return User.create({
    firstName: 'Giovana',
    lastName: 'Aguilar'
  }, {transaction: t}).then(user => {
    return user.addBrother({
      firstName: 'Davi',
      lastName: 'Aguilar'
    }, {transaction: t});   });
}).then(result => {
  // Transação commitada automaticamente
}).catch(err => {
  // Feito rollback automático da transação});
```

Transações

- **Transação Não Gerenciada (*then-callback*):**
  - Requer que o programador controle explicitamente a transação;
  - Commit com o método ***t.commit();***
  - Rollback com o método ***t.rollback().***

```
return sequelize.transaction().then(t => {
  return User.create({
    firstName: 'Gustavo',
    lastName: 'Aguilar'
  }, {transaction: t}).then(user => {
    return user.addWife({
      firstName: 'Juliana',
      lastName: 'Aguilar'
    }, {transaction: t});
  }).then(() => {
    return t.commit();
  }).catch((err) => {
    return t.rollback();   }); });
```

Conclusão

☑ Associações possibilitam a execução de joins (relacionamentos) entre as classes.

☑ Transações podem ser feitas automaticamente pelo Sequelize, ou explicitamente pelo programador.

Próxima aula

- Capítulo 3: Bancos de Dados NoSQL.