Laboratório 7

Sistemas Distribuídos
Prof Adailton de Jesus Cerqueira Junior

Neste laboratório, daremos continuidade à criação de uma aplicação simples utilizando NodeJS com o framework Express. Agora, vamos aplicar um padrão de projeto para isolar a lógica de acesso aos dados, fornecendo uma interface abstrata para operação de persistência sobre a fonte de dados.

Servidor:

1 – Crie uma pasta chamada lab07 e dentro desta pasta vamos copiar os arquivos criados no laboratório 6.

NOTA: *Você pode fazer esse laboratório direto na pasta do lab06, mas aconselho a criação de um novo diretório para manter o histórico dos laboratórios.*

2 – Dentro da pasta lab07, vamos criar duas pastas chamadas daos e models.

3 – A pasta models será responsável por armazenar nossas classes que representam nossos modelos de dados. Nela iremos criar duas classes:

- Jogo.js

```
class Jogo {
    constructor(id, nome, categoria, ano, fkEmpresa) {
        this.id = id;
        this.nome = nome;
        this.categoria = categoria;
        this.ano = ano;
        this.fkEmpresa = fkEmpresa;
    }
}

module.exports = Jogo;
```

- Empresa.js

```
class Empresa {
    constructor(id, nome) {
        this.id = id;
        this.nome = nome;
    }
}

module.exports = Empresa;
```

NOTA: *Dedique um tempo para analisar os modelos de dados propostos.*

4 – A pasta daos será responsável por armazenar nossas classes de acesso aos modelos de dados. Nela iremos criar duas classes: JogoDAO e EmpresaDAO.

Class JogoDAO:

1 - A classe JogoDAO será nosso Data Access Object que irá conter todas as funções de acesso aos registros do jogo no SGBD.

```
const db = require("../db").db;
const Jogo = require("../models/Jogo");

class JogoDAO {

}

module.exports = JogoDAO;
```

2 - Dentro da nossa classe JogoDAO, vamos criar dois métodos de busca. Um irá buscar pela chave primária (id) do registro. E o outro método irá buscar todos os registros podendo aplicar um filtro de categoria.

```
static findAll(categoria, callback) {
    let query = "SELECT * FROM jogos";

    // Verificando se foi passado um parâmetro de busca
    if (categoria) {
        query += " WHERE categoria LIKE '%" + categoria + "%'";
    }

    db.all(query, [], (err, rows) => {
        if (err) return callback(err, null);
        const jogos = rows.map(row => new Jogo(row.id, row.nome, row.categoria, row.ano, row.fk_empresa));
        callback(null, jogos);
    });
}

static findById(id, callback) {
    const query = "SELECT * FROM jogos WHERE id = ?";
    db.get(query, [id], (err, row) => {
        if (err) return callback(err, null);
        if (!row) return callback(null, null);
        callback(null, new Jogo(row.id, row.nome, row.categoria, row.ano, row.fk_empresa));
    });
}
```

3 - Vamos agora criar outros métodos para criar, atualizar e deletar um registro do jogo dentro da nossa base.

```
static create(nome, categoria, ano, fkEmpresa, callback) {
    const query = "INSERT INTO jogos (nome, categoria, ano, fk_empresa) VALUES (?, ?, ?, ?)";
    db.run(query, [nome, categoria, ano, fkEmpresa], function (err) {
        if (err) return callback(err, null);
        callback(null, new Jogo(this.lastID, nome, categoria, ano, fkEmpresa));
    });
}
```

```
static update(id, nome, categoria, ano, callback) {
    const query = "UPDATE jogos set nome = ?, categoria = ?, ano = ? where id = ?";
    db.run(query, [nome, categoria, ano, id], function (err) {
        if (err) return callback(err);
        callback(null, this.changes > 0);
    });
}
```

```
static delete(id, callback) {
    const query = "DELETE FROM jogos WHERE id = ?";
    db.run(query, [id], function (err) {
        if (err) return callback(err);
        callback(null, this.changes > 0);
    });
}
```

NOTA: *Dedique um tempo para analisar o que os métodos estão fazendo.*

Class EmpresaDAO:

1 - A classe EmpresaDAO é análoga à classe JogoDAO. A diferença é que ela tem acesso aos registro da empresa dentro da nossa base de dados.

```
const db = require("../db").db;
const Empresa = require("../models/Empresa");

class EmpresaDAO {

}

module.exports = EmpresaDAO;
```

2 - Dentro da nossa classe EmpresaDAO, vamos criar o método de busca pela chave primária (id) do registro e outro método busca de todos os registros podendo aplicar um filtro de nome.

```
static findAll(nome, callback) {
    let query = "SELECT * FROM empresas";

    // Verificando se foi passado um parâmetro de busca
    if (nome) {
        query += " WHERE nome LIKE '%" + nome + "%'";
    }

    db.all(query, [], (err, rows) => {
        if (err) return callback(err, null);
        const empresas = rows.map(row => new Empresa(row.id, row.nome));
        callback(null, empresas);
    });
}

static findById(id, callback) {
    const query = "SELECT * FROM empresas WHERE id = ?";
    db.get(query, [id], (err, row) => {
        if (err) return callback(err, null);
        if (!row) return callback(null, null);
        callback(null, new Empresa(row.id, row.nome));
    });
}
```

3 - Vamos criar também os métodos para criar, atualizar e deletar um registro da empresa dentro da nossa base.

```
static create(nome, callback) {
    const query = "INSERT INTO empresas (nome) VALUES (?)";
    db.run(query, [nome], function (err) {
        if (err) return callback(err, null);
        callback(null, new Empresa(this.lastID, nome));
    });
}
```

```
static update(id, callback) {
    const query = "UPDATE empresas set nome = ? where id = ?";
    db.run(query, [nome, id], function (err) {
        if (err) return callback(err);
        callback(null, this.changes > 0);
    });
}
```

```
static delete(id, callback) {
    const query = "DELETE FROM empresas WHERE id = ?";
    db.run(query, [id], function (err) {
        if (err) return callback(err);
        callback(null, this.changes > 0);
    });
}
```

NOTA: *Dedique um tempo para analisar o que os métodos estão fazendo.*

Class Database:

1 – Na classe Database, vamos fazer algumas alterações. A primeira será no método *_createTable()*.

```
_createTable() {
  // Criação da tabela empresas
  const tbEmpresa = `
    CREATE TABLE IF NOT EXISTS empresas (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      nome TEXT NOT NULL UNIQUE
    );
  `;
  this.db.run(tbEmpresa, (err) => {
    if (err) console.error("Erro ao criar tabela: ", err.message);
    else {
      console.log("Tabela 'empresas' verificada/criada.");
      this._seed();
    }
  });
```

```
  // Criação da tabela jogos
  const tbJogo = `
    CREATE TABLE IF NOT EXISTS jogos (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      nome TEXT NOT NULL UNIQUE,
      categoria TEXT NOT NULL,
      ano INTEGER NOT NULL,
      fk_empresa INTEGER NOT NULL,
      FOREIGN KEY(fk_empresa) REFERENCES empresas(id)
    );
  `;
  this.db.run(tbJogo, (err) => {
    if (err) console.error("Erro ao criar tabela: ", err.message);
    else console.log("Tabela 'jogos' verificada/criada.");
  });
}
```

ATENÇÃO: Note que a tabela jogos agora tem uma chave estrangeira (foreign key) para a tabela empresa.

2 – Agora vamos criar um método chamado *_seed()*. Ele será responsável por inserir algumas empresas na nossa tabela de empresa.

```
_seed() {
  const query = "INSERT INTO empresas (nome) VALUES (?)";
  this.db.run(query, ["Nintendo"], (err) => {
    if (err) console.error("Erro ao criar empresa: ", err.message);
    else console.log("Empresa criada.");
  });
  this.db.run(query, ["Ubisoft"], (err) => {
    if (err) console.error("Erro ao criar empresa: ", err.message);
    else console.log("Empresa criada.");
  });
  this.db.run(query, ["Dumativa"], (err) => {
    if (err) console.error("Erro ao criar empresa: ", err.message);
    else console.log("Empresa criada.");
  });
  this.db.run(query, ["Bethesda"], (err) => {
    if (err) console.error("Erro ao criar empresa: ", err.message);
    else console.log("Empresa criada.");
  });
}
```

ATENÇÃO: Note que o método *_seed()* foi chamado na criação da tabela de empresas.

NOTA: *Dedique um tempo para analisar o que os métodos estão fazendo.*

Rotas:

1 – Precisaremos realizar algumas modificações no arquivo **index.js** e nas rotas existentes para utilizar o DAO.

2 – Vamos ajustar o início do arquivo **index.js**, incluindo informações, como a importação dos DAOs.

```
const express = require('express');
const JogoDAO = require('./daos/JogoDAO');
const EmpresaDAO = require('./daos/EmpresaDAO');

const app = express();
const APP_PORT = process.env.APP_PORT || 3000;
```

3 – Agora, vamos atualizar as rotas que utilizam o verbo GET, conforme apresentado a seguir.

```
app.get('/jogos', (req, res) => {
  JogoDAO.findAll(req.query.categoria, (err, jogos) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json(jogos);
  });
});

app.get('/jogos/:id', (req, res) => {
  const id = req.params.id;
  JogoDAO.findById(id, (err, jogo) => {
    if (err) return res.status(500).json({ error: err.message });
    if (jogo) {
      res.json(jogo);
    } else {
      res.status(404).json('Jogo não encontrado.');
    }
  });
});
```

NOTA: *Compare a implementação das rotas do laboratório 6 com essa nova implementação. Perceba que agora está encapsulado a forma como os dados são acessados.*

4 – Para finalizar, vamos atualizar os demais endpoints.

POST:

```
app.post('/jogos', (req, res) => {
  const { nome, categoria, ano, fkEmpresa } = req.body;
  if (!nome && !categoria && !ano && !fkEmpresa) return res.status(400)
    .json({ error: "Campos nome, categoria e ano são obrigatórios" });

  JogoDAO.create(nome, categoria, ano, fkEmpresa, (err, jogo) => {
    if (err) return res.status(500).json({ error: err.message });
    res.status(201).json(jogo);
  });
});
```

PUT:

```
app.put('/jogos/:id', (req, res) => {
  const { nome, categoria, ano } = req.body;
  const id = req.params.id;

  JogoDAO.update(id, nome, categoria, ano, (err, jogo) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!jogo) return res.status(404).json({ error: "Jogo não encontrado" });
    res.json(jogo);
  });
});
```

DELETE

```
app.delete('/jogos/:id', (req, res) => {
  const id = req.params.id;

  JogoDAO.delete(id, (err, jogo) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!jogo) return res.status(404).json({ error: "Jogo não encontrado." });
    res.json({ message: "Jogo removido com sucesso." });
  });
});
```

5 – Analise cada uma das modificações realizadas nos endpoints.

6 - Crie agora os endpoints para empresa.