# Laboratório 6

**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 modificar nosso arquivo de persistência por um SGBD (Sistema de Gerenciamento de Banco de Dados) chamado SQLite.

## Servidor:

1 – Crie uma pasta chamada lab06 e dentro desta pasta vamos copiar os arquivos criados no laboratório 5.

**NOTA:** *Você pode fazer esse laboratório direto na pasta do lab05, mas aconselho a criação de um novo diretório para manter o histórico dos laboratórios.*

2 – Agora vamos instalar os pacotes do SQLite com o comando *npm install sqlite3.*

3 – Vamos instalar um outro pacote auxiliar chamado Dotenv com o comando *npm install dotenv*. Esse pacote será utilizado para carregar variáveis de ambiente de um arquivo .env para a aplicação.

## Class Database:

1 – Na pasta do projeto, vamos criar um arquivo chamado **.env** que será utilizado para armazenar as variáveis de ambiente. Neste arquivo vamos adicionar as informações referente ao nome do nosso banco de dados e a porta default que será utilizada pela aplicação.

**ATENÇÃO:** Note que tem um ponto no começo do nome do arquivo.

```
DB_NAME=database.db
APP_PORT=3000
```

2 – Agora vamos criar um arquivo chamado **db.js** que será utilizado para a configuração do banco de dados. Neste arquivo iremos adicionar a importação dos pacotes e criaremos a classe Database.

```
const sqlite3 = require("sqlite3").verbose();
require("dotenv").config();

class Database {

}
```

3 – Dentro da classe Database, vamos criar um método para criar a tabela de jogos que será responsável por armazenar os registros.

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

**NOTA:** *Dedique um tempo para analisar o que esse método está fazendo.*

4 – Agora, vamos criar um método para realizar a conexão com o banco de dados. Aqui será utilizada a variável de ambiente **DB_NAME**.

```
_connect() {
  this.db = new sqlite3.Database(process.env.DB_NAME, (err) => {
    if (err) {
      console.error("Erro ao conectar no SQLite: ", err.message);
    } else {
      console.log("Conectado ao SQLite.");
      this._createTable();
    }
  });
}
```

5 – Agora, vamos criar o método construtor da classe Database.

```
constructor() {
  if (!Database.instance) {
    this._connect();
    Database.instance = this;
  }
  return Database.instance;
}
```

6 – Para finalizar, vamos adicionar a exportação da classe.

```
// Exporta uma única instância do banco
module.exports = new Database();
```

# Rotas:

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

2 – Vamos ajustar o início do arquivo **index.js**, incluindo informações, como a importação do banco de dados.

```
const express = require('express');
const db = require('./db').db; // Importa a instância do banco de dados

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

// Realiza um parse do body para uma estrutura JSON
app.use(express.json());

app.listen(APP_PORT, () => {
  console.log(`API de jogo em execução na porta ${APP_PORT}.`);
  console.log(`Acesse a url http://localhost:${APP_PORT}`);
});

app.get('/', (req, res) => res.send('API Version 1.1.0 on-line!'));
```

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

```
app.get('/jogos', (req, res) => {
  let query = "SELECT * FROM jogos";

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

  db.all(query, [], (err, jogos) => {
    if (err) return res.status(500).json({ error: err.message });
    res.send(jogos);
  });
});
```

```
app.get('/jogos/:id', (req, res) => {
  let query = "SELECT * FROM jogos where id = ?";
  db.get(query, [req.params.id], (err, jogo) => {
    if (err) return res.status(500).json({ error: err.message });
    if (jogo) {
      res.send(jogo);
    } else {
      res.status(404).send('Jogo não encontrado.');
    }
  });
});
```

4 – Para finalizar, vamos atualizar os demais endpoints.

POST:

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

  db.run("INSERT INTO jogos (nome, categoria, ano) VALUES (?, ?, ?)",
      [nome, categoria, ano], function (err) {
    if (err) return res.status(500).json({ error: err.message });
    res.status(201).send({ id: this.lastID, nome });
  });
});
```

PUT:

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

  let query = "SELECT * FROM jogos where id = ?";
  db.get(query, [id], (err, jogo) => {
    if (err) return res.status(500).json({ error: err.message });
    if (jogo) {
      db.run("UPDATE jogos set nome = ?, categoria = ?, ano = ? where id = ?",
          [nome, categoria, ano, id], function (err) {
        if (err) return res.status(500).json({ error: err.message });
        res.send(jogo);
      });
    } else {
      res.status(404).send('Jogo não encontrado.');
    }
  });
});
```

DELETE

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

  db.run("DELETE FROM jogos where id = ?", [id], function (err) {
    if (err) return res.status(500).json({ error: err.message });
    res.send('Jogo removido com sucesso.');
  });
});
```

5 – Analise cada uma das modificações realizadas nos endpoints.