Banco de Dados + Node.js

Prof. Guilherme Henrique de Assis

2022

SUMÁRIO

> Capítulo 1

Capítulo 1. Introdução

Seja bem-vindo ao módulo de Bancos de Dados + Node.js! Esta apostila é um material de complemento das aulas gravadas, que são a principal fonte de conteúdo da disciplina. Utilize a apostila apenas como material de apoio, e não como a única fonte de estudo.

Esta disciplina tem por objetivo discutir a respeito da integração do Node.js com bancos de dados. Apesar de não ser o intuito principal estudar bancos de dados, iniciaremos com uma breve contextualização, os tipos de bancos existentes e suas diferenças.

Em seguida falaremos a respeito de bancos de dados SQL, falando um pouco sobre a linguagem SQL e contextualizando os principais comandos. Falaremos mais especificamente sobre o PostgreSQL, que será o banco de dados relacional utilizado na disciplina. Utilizaremos um serviço de nuvem com cota gratuita para armazenar o banco de dados criado, o ElephantSQL.

Será desenvolvida uma API com Node.js e Express, que será integrada ao banco de dados PostgreSQL, inicialmente utilizando os comandos de forma nativa. Posteriormente o projeto será alterado para a inclusão do Sequelize, um ORM para o Node.js

Em seguida abordaremos os bancos de dados NoSQL, mais especificamente o MongoDB. Também utilizaremos um serviço de nuvem para hospedar uma instância do banco com cota gratuita, chamada MongoDB Atlas. Utilizaremos o MongoDB Compass como ferramenta para realizar nossas consultas. Iremos integrar este banco de dados a API que vêm sendo construída, inicialmente utilizando o driver nativo de conexão do MongoDB, e posteriormente utilizando a biblioteca Mongoose.

1.1. Introdução aos Bancos de Dados

Grande parte das aplicações desenvolvidas atualmente possuem a necessidade de armazenar informações. Podemos armazenar estes dados de diversas formas, seja na memória do próprio computador, em um arquivo (Excel por exemplo), ou em bancos de dados.

Podemos definir bancos de dados como uma coleção organizada de dados. Neles podemos armazenar nossos dados de uma forma que facilite a sua recuperação e armazenamento.

Um banco de dados precisa de um software para servir de interface entre as informações e quem vai manipulá-las, atuando como uma interface entre o banco e os usuários ou programas que estejam utilizando-o. Este software é chamado sistema de gerenciamento de bancos de dados (SGBD), ou em inglês, *data base management system* (DBMS).

Existem bancos de dados relacionais e bancos de dados não relacionais. Nos bancos de dados relacionais os dados são armazenados em tabelas. Nestas tabelas, as colunas representam os atributos e as linhas representam os registros. Estes bancos utilizam a linguagem SQL (Structured Query Language) para a manipulação de dados. O SQL permite a criação de consultas complexas, com relacionamentos entre várias tabelas. Podemos citar como exemplos de bancos de dados relacionais o MySQL, SQL Server, Oracle e PostgreSQL, que utilizaremos neste módulo.

Os bancos de dados não relacionais também são chamados de NoSQL. Esta sigla quer dizer “not only SQL”. Eles permitem trabalhar com dados semiestruturados (logs, imagens etc.), sendo na maioria das vezes mais facilmente escaláveis que os bancos de dados relacionais. Dentro da categoria de bancos não relacionais, temos vários tipos diferentes de bancos, que possuem características e finalidades diferentes. Segue abaixo alguns deles:

- Orientado a documentos: os documentos são coleções de atributos e valores, não precisam ter uma estrutura em comum. Exemplo: MongoDB;

- Chave-valor: trabalha com tabelas hash, no qual os objetos são indexados pelas suas chaves. Exemplo: Redis;

- Orientados a colunas: armazenam os registros em disco de acordo com as colunas, facilitando a leitura. Exemplo: Cassandra;

- Orientados a grafos: armazenam os dados no formato de grafos, no qual os relacionamentos são muito importantes. Exemplo: Neo4J.

> Capítulo 2

Capítulo 2. Bancos de Dados SQL

2.1. Bancos de Dados SQL

Nos bancos de dados relacionais (ou SQL), os registros são armazenados em tabelas. Um mesmo banco pode ter várias tabelas. Geralmente os desenvolvedores criam uma tabela para cada domínio relacionado a regra de negócio. Por exemplo, em um sistema de um e-commerce, poderia existir tabelas para armazenar produtos, fornecedores, vendas etc.

Nas tabelas as colunas indicam as propriedades enquanto as linhas representam os registros. Em um cadastro de clientes por exemplo, as colunas poderiam ser “nome”, “cpf” e “telefone”. Uma linha representaria os dados de um cliente, como “João”, “12345678911” e “(99)99999-9999”. A imagem abaixo ilustra uma tabela com suas colunas e linhas.

Figura 1 – Tabela de banco de dados.

client_id	name	cpf	phone	email	address
1	João Silva	95291452099	31998765432	joao.silva@gmail.com	Rua X, 123
2	Maria Aparecida	94424615081	31988795462	maria.aparecida@gmail.com	Rua Y, 321

Ao criar uma tabela em um banco relacional, o desenvolvedor precisa definir o tipo de dados que cada uma das colunas terá. Por exemplo, se a coluna aceitará números, texto, booleanos etc.

Um aspecto importante a se definir nas tabelas de bancos relacionais é a sua chave primária. A chave primária geralmente trata-se de uma coluna, que será utilizada para identificar cada registro de forma única. A chave primária de uma tabela não pode ser repetida e também não pode ser nula. A partir dela é possível acessar um registro em específico, sem a possibilidade de ocorrerem duplicidades.

Por exemplo, em uma tabela que armazena clientes, poderíamos definir a coluna “cliente_id” como chave primária desta tabela, permitindo que ela recebesse valores inteiros. Caso nesta tabela já possua um registro com “client_id” igual a 2 por exemplo, o próprio banco de dados bloquearia uma tentativa de inserção de outro registro com o mesmo identificador (id).

2.2. PostgreSQL

O PostgreSQL é um SGBD open-source, criado em 1985. Ele pode ser utilizado no Windows, Linux e MacOS. Ele utiliza a linguagem SQL para manipulação de dados. Ele aceita vários tipos de dados, como por exemplo: INT, VARCHAR, DATE, TIMESTAMP, NUMERIC, entre outros.

Figura 2 – PostgreSQL.

Fonte: https://www.postgresql.org/.

É possível realizar o download do PostgreSQL e instalá-lo localmente na máquina. Porém, nas aulas gravadas é utilizado um serviço em nuvem do PostgreSQL, chamado ElephantSQL (https://www.elephantsql.com/). Ele possui uma cota gratuita de utilização, suficiente para a realização dos testes.

Figura 3 – ElephantSQL.

Fonte: https://www.elephantsql.com/.

2.3. Comandos SQL

Existem vários tipos de comandos SQL com os quais podemos manipular informações no banco de dados. Por serem comandos da linguagem SQL, eles funcionam de forma muito parecida em todos os bancos de dados que adotam essa linguagem. Abaixo falaremos brevemente sobre alguns dos principais comandos, sua finalidade e colocaremos uma imagem com um exemplo do comando.

- Create Table: utilizado para criar uma tabela no banco de dados.

Figura 4 – Create Table.

```
CREATE TABLE clients (
  client_id SERIAL PRIMARY KEY,
  name VARCHAR NOT NULL,
  cpf VARCHAR NOT NULL,
  phone VARCHAR NOT NULL,
  email VARCHAR NOT NULL,
  address VARCHAR NOT NULL
)
```

- Insert into: utilizado para inserir registros na tabela.

Figura 5 – Insert Into.

```
INSERT INTO clients (
  name, cpf, phone, email, address
)
VALUES (
  'João Silva', '95291452099', '31998765432',
  'joao.silva@gmail.com', 'Rua X, 123'
)
```

- Select: utilizado para consultar registros na tabela. Na cláusula “SELECT” são inseridos os campos que deseja que sejam retornados, no “FROM” qual tabela está sendo buscado os dados e no “WHERE” os filtros desejados.

Figura 6 – Select.

```
SELECT * FROM clients


SELECT name, email
  FROM clients
 WHERE client_id = 1
```

- Update: utilizado para atualizar registros na tabela.

Figura 7 – Update.

```
UPDATE clients
   SET phone = '31998765433',
       address = 'Rua X, 124'
 WHERE client_id = 1
```

- Delete: utilizado para excluir registros da tabela.

Figura 8 – Delete.

```
DELETE
  FROM clients
 WHERE client_id = 2
```

2.4. Integrando Node.js ao PostgreSQL

Uma das formas de integrar o Node.js ao PostgreSQL é utilizando o seu driver nativo de conexão. Para isso, basta realizar a instalação da biblioteca “pg” no projeto, utilizando o NPM através do comando “npm install pg”. A imagem abaixo ilustra uma função que utiliza esta biblioteca para conectar a um banco de dados.

Figura 9 – Conexão ao PostgreSQL.

Fonte: Elaborado pelo próprio professor.

Na figura acima podemos reparar que a conexão ao banco é definida a partir de uma string de conexão. Essa string pode ser recuperada no servidor configurado, no nosso caso no ElephantSQL. Ela irá conter o endereço do banco com usuário e senha que será utilizado para a conexão.

Uma observação importante nesse código se trata do pool de conexões. O pool de conexões é uma forma que o banco de dados utiliza para gerenciar várias conexões. Um banco de dados consegue gerenciar um certo limite de conexões simultâneas, dessa forma abrir conexões indiscriminadamente pode acabar prejudicando a performance, até mesmo tornando o serviço indisponível.

Porém, o processo de abrir e fechar uma conexão a cada requisição tem um certo custo, e em aplicações com alto fluxo isso pode gerar problemas. Por isso os bancos de dados trabalham com o conceito de pool de conexões. A partir dele o banco mantém algumas conexões abertas mesmo após não estarem mais sendo utilizadas, e quando alguém solicita

uma nova conexão ele reutiliza as que já estão abertas, agilizando assim o processo.

Após criar o código responsável por conectar ao banco de dados, para executar um comando SQL no banco de dados basta requisitar uma nova conexão e realizar a consulta. No caso do driver nativo do MongoDB para o Node.js, as consultas e alterações são realizadas a partir da função “query”. A figura abaixo ilustra a execução de um comando de inserção com passagem de parâmetros.

Figura 10 – Execução de comandos com parâmetros.

```
async function insertClient(client) {
    const conn = await connect();
    try {
        const sql =
            "INSERT INTO clients (name, cpf, phone, email, address) " +
            "VALUES ($1, $2, $3, $4, $5) RETURNING *";
        const values = [client.name, client.cpf, client.phone,
            client.email, client.address];
        const res = await conn.query(sql, values);
        return res.rows[0];
    } catch (err) {
        throw err;
    } finally {
        conn.release();
    }
}
```

Uma observação importante a respeito do código acima é a forma como os parâmetros são passados. Os parâmetros são colocados dentro do comando inicialmente com o símbolo $ e um número indicando a ordem com que serão substituídos, começando em 1. Na função “query”, o primeiro parâmetro indica o comando SQL que será executado, e o segundo consistem uma lista com os parâmetros que irão substituir os símbolos $ no momento da execução. A responsabilidade de realizar essa substituição é da própria biblioteca do PostgreSQL.

Concatenar os parâmetros manualmente no comando, utilizando concatenação de strings não é indicado, pois deixa o código vulnerável a ataques do tipo SQL Injection. Estes ataques ocorrem quando usuários mal-intencionados digitam comandos SQL no local das informações corretas.

Caso o código esteja concatenando essas informações diretamente no comando, esses comandos indevidos podem ser executados diretamente no banco de dados, realizando ações indesejadas. Ao passar parâmetros da maneira adequada, a própria biblioteca se responsabiliza por evitar que estes parâmetros sejam executados como consultas, garantindo uma segurança maior a aplicação.

A figura abaixo ilustra uma consulta que pode retornar vários resultados. Estes resultados são obtidos a partir da propriedade “rows”. Outro detalhe importante é a cláusula finally do try catch, que garante que a conexão seja liberada sempre após a finalização do SQL, independentemente se ela foi executada com sucesso ou retornou um erro.

O fato de chamar o “release” não indica que a conexão será encerrada, isso fica a cargo do pool de conexões, pois ele é quem decidirá se irá reutilizar esta conexão ou não. O “release” é importante pois ele informa ao pool que esta conexão não será mais utilizada.

Figura 11 – Execução de consultas.

```
async function getSales() {
    const conn = await connect();
    try {
        const res = await conn.query("SELECT * FROM sales");
        return res.rows;
    } catch (err) {
        throw err;
    } finally {
        conn.release();
    }
}
```

Em um projeto organizado em camadas, os códigos responsáveis por interagir com o banco de dados devem estar localizados na camada “repository”, pois a responsabilidade da persistência de dados é dela. Manter todo o código que interage com o banco de dados em um pacote separado é muito bom para isolar as demais partes. Dessa forma, a aplicação pode até mesmo trocar de banco de dados que as demais camadas não serão afetadas, desde que as assinaturas das funções permaneçam as mesmas.

> Capítulo 3

Capítulo 3. Bancos de Dados SQL com ORM

3.1. Sequelize

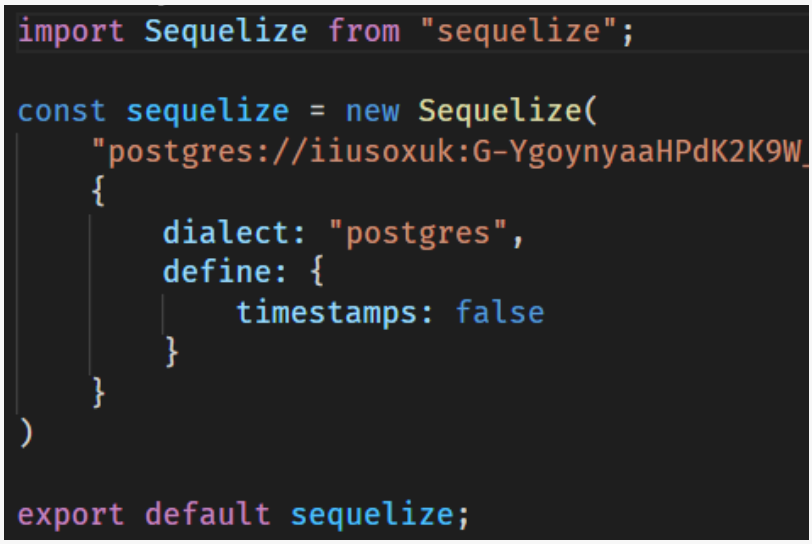
ORM é a sigla para Object-Relational Mappers. Bibliotecas ORM são utilizadas para realizar o mapeamento das entidades para classes, evitando assim código repetitivo, como por exemplo comandos SQL. Eles mapeiam tabelas para classes, facilitando algumas partes do desenvolvimento. O Sequelize é uma biblioteca ORM para o Node.js. Ela possui implementações para os bancos de dados PostgreSQL, MySQL, MariaDB, SQLite e SQLServer.

Figura 12 – Sequelize.

Fonte: https://sequelize.org/.

A figura abaixo ilustra um trecho de código que realiza a conexão com o banco de dados utilizando o Sequelize. É criado o objeto do Sequelize, passando como parâmetro qual o tipo de bancos de dados que ele irá conectar. O segundo parâmetro que trata sobre timestamps, informa ao Sequelize para ele não criar por padrão os campos “createdAt” e “updatedAt”, com as datas de criação e atualização, em todas as tabelas que ele mapear. Esse é um comportamento padrão que pode ser alterado.

Figura 13 – Conexão com Sequelize.



A figura abaixo ilustra o mapeamento de uma entidade com o Sequelize. A partir do mapeamento, as tabelas e propriedades ficam associadas a uma classe, facilitando assim a manipulação dos dados. Enviar e buscar dados ao banco passa a ser feito utilizando as propriedades mapeadas, e não mais parâmetros de acordo com a posição. Esta abordagem diminui a quantidade de código a ser escrito no momento de interagir com o banco de dados.

Porém, em algumas consultas mais complexas, pode ficar bastante complicado realizá-la com o Sequelize. Nestas situações é possível executar a consulta SQL de forma nativa.

Figura 14 – Mapeamento com Sequelize.

```
import Sequelize from "sequelize";
import db from "../repositories/db.js";

const Client = db.define('clients', {
    clientId: {
        type: Sequelize.INTEGER,
        autoIncrement: true,
        allowNull: false,
        primaryKey: true
    },
    name: {
        type: Sequelize.STRING,
        allowNull: false
    },
    cpf: {
        type: Sequelize.STRING,
        allowNull: false
    },
    phone: {
        type: Sequelize.STRING,
        allowNull: false
    },
    email: {
        type: Sequelize.STRING,
        allowNull: false
    },
    address: {
        type: Sequelize.STRING,
        allowNull: false
    }
}, { underscored: true });

export default Client;
```

No mapeamento realizado no trecho de código acima podemos notar que o primeiro parâmetro passado para a função “define” é o nome da tabela correspondente no banco de dados. Posteriormente são definidos todos os campos da tabela, com o seu tipo de dados correspondente. Existem outras formas de definir uma entidade com o Sequelize, na documentação é possível ver com mais detalhes (https://sequelize.org/master/manual/model-basics.html).

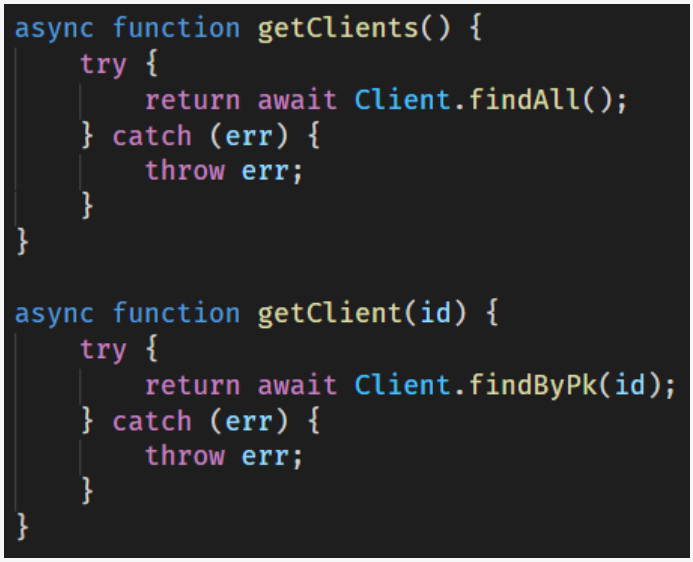
O Sequelize suporta vários tipos de dados, como por exemplo STRING, TEXT, BOOLEAN, INTEGER, DOUBLE etc. A lista completa pode ser encontrada no seguinte link da documentação (https://sequelize.org/master/manual/model-basics.html#data-types).

Os modelos mapeados podem ser separados em uma camada durante a organização do projeto. O pacote que armazena os modelos mapeados pode ser chamado de “models”, por exemplo.

A figura abaixo ilustra a utilização de um modelo mapeado para realizar consultas ao banco de dados. A classe é importada, e a partir dela pode ser utilizado diversos comandos já prontos que são fornecidos como o Sequelize.

No primeiro trecho de código da imagem consta uma consulta com a função “findAll”, que irá buscar todos os registros da tabela. O segundo trecho consiste em um “findByPk”, que busca na definição do modelo a chave primária da tabela, e filtra de acordo com o parâmetro fornecido. Para consultar outras funções para consulta de informações, segue o link da documentação (https://sequelize.org/master/manual/model-querying-finders.html).

Figura 15 – Consulta no Sequelize.



A figura abaixo ilustra a manipulação de dados com o Sequelize. A primeira função realiza a criação de um registro, enviando como parâmetro um objeto com as informações de acordo com o modelo, e chamando a função “create”. A segunda função realiza a exclusão de um registro utilizando a função “delete”.

A cláusula “where” é utilizada para filtrar os registros que serão excluídos. A terceira função realiza a atualização de um registro, atualizando as propriedades enviadas como parâmetro, e respeito o filtro definido no “where”. Nesta função não é obrigatório passar todos os campos para serem atualizados, pode ser enviado somente os campos que sofrerão alterações.

Figura 16 – Manipulação de dados no Sequelize.

```
async function insertClient(client) {
    try {
        return await Client.create(client);
    } catch (err) {
        throw err;
    }
}

async function deleteClient(id) {
    try {
        await Client.destroy({
            where: {
                clientId: id
            }
        });
    } catch (err) {
        throw err;
    }
}

async function updateClient(client) {
    try {
        await Client.update(client, {
            where: {
                clientId: client.clientId
            }
        });
        return await getClient(client.clientId);
    } catch (err) {
        throw err;
    }
}
```

Ao utilizar o Sequelize, não é preciso se preocupar com o pool de conexões, nem com o fechamento das conexões. É o próprio Sequelize quem se encarrega de fazer este gerenciamento.

> Capítulo 4

Capítulo 4. Bancos de Dados NoSQL

Os bancos de dados relacionais são preocupados com a consistência e relacionamento das informações, porém quando é necessária uma alta escalabilidade, toda esta organização pode acabar prejudicando. Neste cenário surgem os bancos de dados NoQL (Not Only SQL), também chamados de não relacionais.

4.1. MongoDB

O MongoDB é um banco de dados orientado a documentos. Nele não há a definição de um schema fixo, atuando de forma não relacional. Isso permite que documentos de uma mesma coleção possuam colunas diferentes. O MongoDB armazena seus documentos em estruturas JSON, facilitando assim a manipulação com o Node.js.

Figura 17 – MongoDB.

Fonte: https://www.mongodb.com/.

Realizando uma analogia da estrutura de um banco SQL com um NoSQL, podemos relacionar os seguintes itens:

SQL	NoSQL
Bancos de Dados	Bancos de Dados
Tabelas	Coleções
Linhas	Documentos
Colunas	Campos

A imagem abaixo ilustra como é realizada a conexão ao MongoDB utilizando o driver nativo para conexão.

Figura 18 – Conexão ao MongoDB.

```
import mongodb from "mongodb";

function getClient() {
    const uri = "mongodb+srv://root:igti@
    return new mongodb.MongoClient(uri);
}

export { getClient }
```

No MongoDB os dados ficam salvos em coleções, que podemos fazer uma analogia com as tabelas nos bancos relacionais. Cada registro é um documento, sendo ele um objeto com estrutura JSON. A vantagem destes bancos é que todos os dados relacionados a este documento podem ficar salvos juntos dentro do mesmo documento.

Por exemplo, em um registro de uma venda, os dados de cliente, fornecedor, produto etc., poderiam ser propriedades deste documento da venda, sendo representados por um objeto JSON dentro deste documento. Isso facilita a recuperação de um grande volume de dados, pois dispensa a necessidade de realizar vários “joins” entre tabelas diferentes, igual seria nos bancos relacionais.

Em contrapartida, as estruturas passam a não ser mais rígidas, fazendo com que documentos diferentes de uma mesma coleção possam ter propriedades diferentes. Outra situação que pode ocorrer é a inconsistências dos valores, já que muitas informações podem estar duplicadas em vários documentos.

A imagem abaixo ilustra a inserção e a atualização de um documento em uma coleção. Primeiro é selecionado o banco de dados desejado, depois é selecionada a coleção, e por fim é chamado o método desejado. Na documentação do MongoDB é possível verificar os diversos tipos de operações possíveis de serem realizadas (https://docs.mongodb.com/drivers/node/current/usage-examples/).

Figura 19 – Criação e atualização de documentos no MongoDB.

```
async function createProductInfo(productInfo) {
    const client = getClient();
    try {
        await client.connect();
        await client.db("store").collection("productInfo").insertOne(productInfo);
    } catch (err) {
        throw err;
    } finally {
        await client.close();
    }
}

async function updateProductInfo(productInfo) {
    const client = getClient();
    try {
        await client.connect();
        await client.db("store").collection("productInfo").updateOne(
            { productId: productInfo.productId },
            { $set: { ...productInfo } }
        );
    } catch (err) {
        throw err;
    } finally {
        await client.close();
    }
}
```

A imagem abaixo ilustra a realização de consultas no MongoDB. A primeira função busca um registro com a função “findOne”. Ela irá retornar o primeiro registro encontrado com o filtro passado como parâmetro. Os filtros no MongoDB são objetos JSON com as propriedades e valores

desejados. O MongoDB utiliza esse filtro para encontrar documentos que possuam essas propriedades e valores na coleção buscada.

A segunda função busca vários registros, pois utiliza a função “find” no lugar do “findOne”, retornando assim todos os documentos que forem encontrados. Neste exemplo o filtro passado está vazio, dessa forma ele retornará todos os documentos. Nos casos em que vários registros serão retornados, é preciso colocar um “toArray()” no final do comando, para que os resultados sejam convertidos para um array, sendo assim possíveis de serem manipulados no JavaScript.

Figura 20 – Consulta de documentos no MongoDB.

```
async function getProductInfo(productId) {
    const client = getClient();
    try {
        await client.connect();
        return await client.db("store").collection("productInfo").findOne({ productId });
    } catch (err) {
        throw err;
    } finally {
        await client.close();
    }
}

async function getProductsInfo() {
    const client = getClient();
    try {
        await client.connect();
        return await client.db("store").collection("productInfo").find({}).toArray();
    } catch (err) {
        throw err;
    } finally {
        await client.close();
    }
}
```

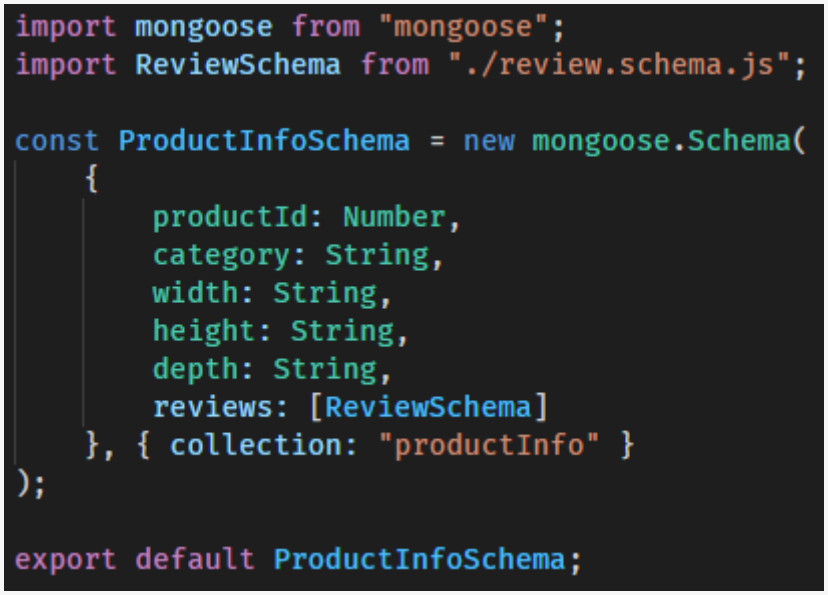
4.2. Mongoose

O Mongoose é uma biblioteca ODM do Node.js para o MongoDB. ODM é a sigla para Object Data Model. É uma biblioteca utilizada para validar as estruturas dos documentos.

Para utilizar o Mongoose é preciso definir um esquema (schema) para cada uma das coleções desejadas. A partir dessa definição, todas as

manipulações dos dados devem respeitar as regras criadas. A imagem abaixo ilustra a definição um schema como exemplo.

Figura 21 – Schema no Mongoose.



Com o schema em mãos, para poder manipular os dados, é preciso criar um model a partir desse schema. Este model é criado a partir da chamada da função “model” da própria biblioteca do Mongoose, passando o schema como parâmetro. Com o objeto do model é possível realizar todas as manipulações desejadas, como inserções, alterações, exclusões e consultas.

A imagem abaixo ilustra um exemplo de uma inserção, depois uma atualização, depois a consulta de um registro filtrando por uma propriedade e por fim uma consulta retornando todos os documentos.

Figura 22 – Operações com o Mongoose.

```
async function createProductInfo(productInfo) {
    try {
        const mongoose = await connect();
        const ProductInfo = mongoose.model("ProductInfo", ProductInfoSchema);
        productInfo = new ProductInfo(productInfo);
        await productInfo.save();
    } catch (err) {
        throw err;
    }
}

async function updateProductInfo(productInfo) {
    try {
        const mongoose = await connect();
        const ProductInfo = mongoose.model("ProductInfo", ProductInfoSchema);
        await ProductInfo.findOneAndUpdate({ productId: productInfo.productId }, productInfo);
    } catch (err) {
        throw err;
    }
}

async function getProductInfo(productId) {
    try {
        const mongoose = await connect();
        const ProductInfo = mongoose.model("ProductInfo", ProductInfoSchema);
        const query = ProductInfo.findOne({ productId });
        return await query.exec();
    } catch (err) {
        throw err;
    }
}


async function getProductsInfo() {
    try {
        const mongoose = await connect();
        const ProductInfo = mongoose.model("ProductInfo", ProductInfoSchema);
        const query = ProductInfo.find({});
        return await query.exec();
    } catch (err) {
        throw err;
    }
}
```

Referências

ELEPHANT SQL. PostgreSQL as a Service – Perfectly configured and optimized PostgreSQL databases in 2 minutes. [*S.l.*], 2021. Disponível em: https://www.elephantsql.com/. Acesso em 18 mar. 2022.

EXPRESS.JS. Express - framework de aplicativo da web Node.js. [*S.l.*], 2021. Disponível em: https://expressjs.com/pt-br/. Acesso em: 18 mar. 2022.

KONG INC. Insomnia – Download page. [San Francisco], 2021. Disponível em: https://insomnia.rest/download. Acesso em: 18 mar. 2022.

MONGODB INC. MongoDB Atlas. New York, 2021. Disponível em: https://www.mongodb.com/cloud/atlas. Acesso em: 18 mar. 2022.

MONGODB INC. MongoDB Compass. New York, 2021. Disponível em: https://www.mongodb.com/pt-br/products/compass. Acesso em: 18 mar. 2022.

MONGODB INC. MongoDB -The database for modern applications. New York, 2021. Disponível em: https://www.mongodb.com/. Acesso em: 18 mar. 2022.

MONGODB INC. Mongoose ODM. Versão v5.12.14, New York, 2021. Disponível em: https://mongoosejs.com/. Acesso em: 18 mar. 2022.

OPENJS FOUNDATION. Node.js. [San Francisco], 2021. Disponível em: https://nodejs.org/en/. Acesso em: 18 mar. 2022.

POSTGRESQL. PostgreSQL: The world's most advanced open source relational database. [*S.l.*], 2021. Disponível em: https://www.postgresql.org/. Acesso em: 18 mar. 2022.

SEQUELIZE. Sequelize ORM. [*S.l.*], 2021. Disponível em: https://sequelize.org/. Acesso em: 18 mar. 2022.