Desenvolver aplicações Backend para WEB

Prof. Renan Ponick

Senac

Lema

“Espera-se que você duvide do que sabe, tenha curiosidade a respeito do que não sabe e atualize suas opiniões diante de novos dados..”

Adam Grant

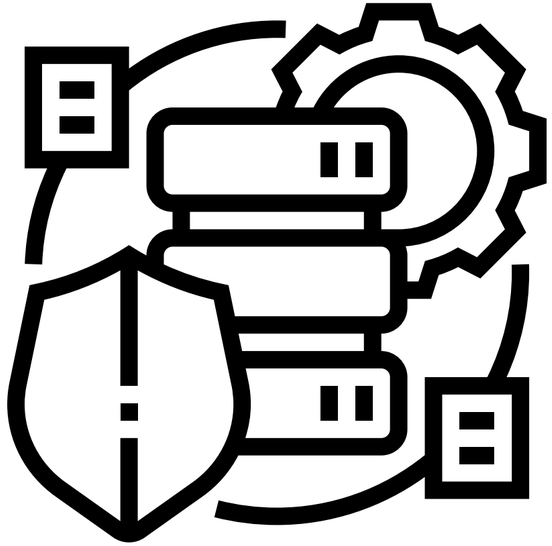
Agenda

28/09/2023 - ORM - Async - Await
29/09/2023 - Exercícios + Testes
04/10/2023 - Revisão
05/10/2023 - ORM - Relações
06/10/2023 - Avaliação
11/10/2023 - Autenticação
12/10/2023 - FERIADO
13/10/2023 - FERIADO
18/10/2023 - Desafios
19/10/2023 - Avaliação
20/10/2023 - Recuperação/Revisão - **ACABOU**

Transactions

Transactions

Uma **transação** é uma sequência de operações que são executadas como uma única unidade, garantindo que ou todas as operações são realizadas com sucesso ou nenhuma delas é realizada.
É uma maneira de garantir a consistência dos dados em um banco de dados

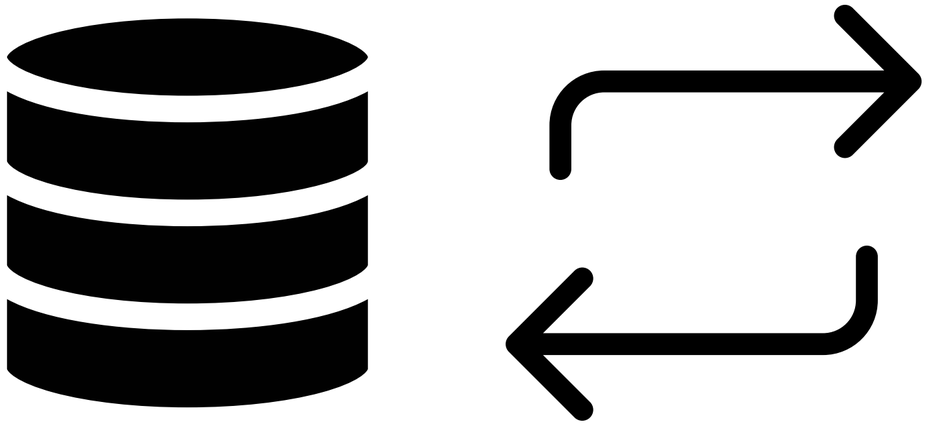


Consistencia

Se você está transferindo dinheiro de uma conta para outra, você quer garantir que o dinheiro seja retirado de uma conta e adicionado à outra. Se alguma parte desse processo falhar (por exemplo, se o dinheiro for retirado, mas não for adicionado à outra conta), isso poderia resultar em um estado inconsistente.

Implícitas

Cada comando que executamos por padrão já é uma transação isolada, quando executamos um comando de INSERT em uma tabela, a inserção só é realmente salva no banco de dados se todo o comando estiver correto.



Explicitas

Ocorre quando nós indicamos a partir de onde começa esse conjunto de comandos que estarão nessa transação

```
BEGIN TRANSACTION
```

```
START TRANSACTION
```

que, por fim, ou tudo é salvo no banco de dados ou nada

```
COMMIT;
```

```
ROLLBACK;
```

Usaremos onde?

Inicialmente usaremos nos testes, para que não sejam gerados/alterados dados do banco atual na nossa máquina.

```
beforeAll(async () => {
    this.transaction = await conexao.transaction()
})
afterAll(async () => {
    this.transaction.rollback()
})
```

```
const result = await servico.Add({
    nome: 'joao',
    email: 'teste2@teste.com',
    senha: '123456'
}, this.transaction)
```

Propagar

Precisamos propagar para o serviço e para o repositório.

```
async Adicionar(pessoa, transaction){
  if(!pessoa) {
    throw new Error("Favor preencher o pessoa.")
  } else if(!pessoa.nome) {
    throw new Error("Favor preencher o nome.")
  } else if(!pessoa.email) {
    throw new Error("Favor preencher o email.")
  } else if(!pessoa.senha) {
    throw new Error("Favor preencher o senha.")
  }

  return repositorio.Adicionar(pessoa, transaction)
}
```

```
async Adicionar(pessoa, transaction){
    return Pessoa.create(pessoa, {
        transaction
    })
}
```

Tabela relacional

Tabela relacional

Na sua criação ela precisará definir a chave relacional da tabela respectiva.

```
const Pessoa = require('./pessoa.js');
const { DataTypes } = require('sequelize');
const sequelize = require('../database.js');

const Cachorro = sequelize.define('Cachorro', {
    nome: {
      type: DataTypes.STRING,
      allowNull: false
    },
    raca: {
      type: DataTypes.STRING
    },
    pessoaId: {
        field: 'pessoa_id',
        type: DataTypes.INTEGER,
        references: {
            model: Pessoa,
            key: 'id'
        }
    }
  }, {
    createdAt: false,
    updatedAt: false,
});

module.exports = Cachorro;
```

Tabela relacional

E precisará ser explicitado essa relação na tabela pai com belongsTo e hasMany

Para mais info, segue a doc:

Associations

Sequelize supports the standard associations: One-To-One, One-To-Many and Many-To-Many.

sequelize.org

```
const Cachorro = require('./cachorro.js');
const { DataTypes } = require('sequelize');
const sequelize = require('../database.js');

const Pessoa = sequelize.define('pessoas', {      You, 2
    id: {
        primaryKey: true,
        type: DataTypes.INTEGER
    },
    nome: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    email: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true,
    },
    senha: {
        type: DataTypes.STRING,
        allowNull: false,
    },
}, {
    createdAt: false,
    updatedAt: false
});

Cachorro.belongsTo(Pessoa, { foreignKey: 'pessoaId' });
Pessoa.hasMany(Cachorro, { foreignKey: 'pessoaId' });

module.exports = Pessoa;
```

Tabela relacional

Para buscar os valores dessa tabela relacional juntamente com a tabela primaria é necessário informar na query que a tabela relacional será inclusa no select

```
async PegarUm(id){
    return Pessoa.findOne({
        where: {
            id
        },
        include: ['Cachorros']
    })
}
```

Até aqui tranquilo?

Exercício parte 2

Crie uma tabela nova dentro do banco PetShop
A classe será o nosso querido - Cachorros
Faça com que a classe tenha relação com a classe Cliente, onde
Um Cachorro pertence a Um Cliente
E
Um Cliente pode ter Vários Cachorros

Feito isso, realize o CRUD do Cachorro.
Faça um metodo para listar todos os cachorros de um Cliente

Exercício parte 1

Crie um repositório no GitHub de nome APIPetShopNodeJs, clone e Inicialize um projeto NOVO.

Instale as dependencias que vimos até aqui e crie a estrutura de pastas corretamente.

Crie um banco de dados novo chamado PetShop, que contenha uma tabela chamada **Clientes,** armazene nesta tabela, id, nome, telefone (único).

Realize o CRUD deste cliente, assim como fizemos com a Pessoa