Desenvolvimento Backend

Aula 07 - Banco de dados relacional (parte 2)

Material Didático do Instituto Metrópole Digital - IMD

Termo de uso

Apresentação

Nesta aula iremos ver como criar relacionamento entre modelos no banco de dados.

Objetivos

- Conhecer como criar dados iniciais no banco de dados
- Conhecer como criar relacionamento entre modelos

Organizando o projeto

Link do video da aula: https://youtu.be/M1TeVE_n6Zk

Inicialmente, é importante lembrar que foi deixada como exercício a implementação da rota de usuários. O código implementado deve ficar de acordo com o código abaixo:

```
const express = require('express')
const router = express.Router()
const validarUsuario =
require('../middleware/validarUsuario.middleware')
const { Usuario } = require('../db/models')

router.post('/', validarUsuario)
router.put('/', validarUsuario)

router.get('/', async (req, res) => {
  const usuarios = await Usuario.findAll()
  res.json({ usuarios: usuarios })
})

router.get('/:id', async (req, res) => {
  const usuario = await Usuario.findByPk(req.params.id)
  if (usuario){
    res.json({ usuario: usuario })
  }else{
    res.status(400).json({msg: "Usuário não encontrado!"})
  }
})

router.post('/', async (req, res) => {
  const usuario = await Usuario.create(req.body)
```

```
  res.json({ msg: "Usuário adicionado com sucesso!", userId:
usuario.id })
})

router.put('/', async (req, res) => {
  const id = req.query.id
  const usuario = await Usuario.findByPk(id)
  if (usuario){
    usuario.email = req.body.email
    usuario.senha = req.body.senha
    await usuario.save()
    res.json({ msg: "Usuário atualizado com sucesso!" })
  }else{
    res.status(400).json({msg: "Usuário não encontrado!"})
  }
})

router.delete('/', async (req, res) => {

  const id = req.query.id
  const usuario = await Usuario.findByPk(id)

  if (usuario){
    try{
      await usuario.destroy()
      res.json({msg: "Usuário deletado com sucesso!"})
    }catch (error){
      res.status(500).json({msg: "Falha ao remover usuário"})
    }
  }else{
    res.status(400).json({msg: "Usuário não encontrado!"})
  }

})

module.exports = router;
```

Organizando as pastas

Visando trabalhos futuros, quando pode ser necessária a aplicação de uma ação em todo o projeto, é preciso criar uma pasta *scr* na raiz do projeto para colocar todo o código dentro dela. Acompanhe a aula e faça as alterações necessárias.

Configurando o Sequelize

Até o momento, toda a configuração do sequelize ficou em uma pasta, porém esse não é o padrão. Então, para uma melhor prática, crie um arquivo *.sequelizerc.js* (arquivo oculto) e adicione a ele as configurações do banco de dados. Ao final, seu arquivo deve ficar como abaixo:

```
    const { resolve } = require('path')

    module.exports = {
            'config': resolve(__dirname, 'src', 'db', 'config',
'config.json'),
            'models-path': resolve(__dirname, 'src', 'db',
'models'),
            'migrations-path': resolve(__dirname, 'src', 'db',
'migrations'),
            'seeders-path': resolve(__dirname, 'src', 'db',
'seeders'),
    }
```

Inserindo dados iniciais com Seeds

Link do video da aula: https://youtu.be/s2p-wKY4w70

Muitas vezes a aplicação precisa que já exista um dado inicial na tabela para funcionar adequadamente.

Criando o arquivo e adicionando os dados.

Para adicionar esses dados iniciais o *Sequelize* dispõe de uma ferramenta chamada *seeders*. Para utilizá-la, é necessário criar o arquivo base, rodando no terminal o seguinte comando:

```
    npx sequelize-cli seed:generate --name root-user
```

Após criado o arquivo, é necessário modificá-lo. Acompanhe a aula e ao final seu arquivo deve conter o código abaixo:

```
    'use strict';

    module.exports = {
```

```
        up: async (queryInterface, Sequelize) => {
         await queryInterface.bulkInsert('Usuarios', [{
             email: 'root@gmail.com',
             senha: 'd8fy83uu4j',
             createdAt: new Date(),
             updatedAt: new Date()
         }])
        },

        down: async (queryInterface, Sequelize) => {
            await queryInterface.bulkDelete('Usuarios', {email:
'root@gmail.com'}, {})
        }
    };
```

Testando as sementes

Para fazer o teste de execução de todos os códigos de sementes e reverter a operação, é necessário rodar no terminal os seguintes comandos:

Executar os códigos de sementes:

```
    npx sequelize-cli db:seed:all
```

Reverter a operação:

```
    npx sequelize-cli db:seed:undo
```

Adicionando relacionamento

Link do video da aula: https://youtu.be/WNBi2OBVJTo

Na aplicação existem dois modelos e, para melhor funcionamento, é necessário estabelecer uma relação entre eles. Existem diversos tipos de relação e deve-se escolher o que mais for adequado para a situação. Para saber mais, acesse o site do Sequelize.

Para esse caso, o tipo de relação usada será a One-To-One.

Estabelecendo a relação

Para relacionar a tabela de post com a de usuários, é necessário implementar a função estática *associate*, no arquivo *post.js*, na pasta *models*. Ademais, é preciso fazer modificações na inicialização do modelo, adicionando uma chave. Após acompanhar o vídeo, seu código deverá estar assim:

```
    'use strict';
    const {
        Model, INTEGER
    } = require('sequelize');
    module.exports = (sequelize, DataTypes) => {
        class Post extends Model {
            /**
             * Helper method for defining associations.
             * This method is not a part of Sequelize lifecycle.
             * The `models/index` file will call this method
automatically.
             */
            static associate(models) {
                Post.belongsTo(models.Usuario, {foreignKey:
'userId'})
            }

        };
        Post.init({
            titulo: DataTypes.STRING,
            texto: DataTypes.STRING,
            userId: DataTypes.INTEGER
        }, {
            sequelize,
            modelName: 'Post',
        });
        return Post;
    };
```

Realizando a migração

Após a implementação de *post.js*, o campo já estará registrado. No entanto, ainda não estará cadastrado no banco, tendo em vista que, quando criado, o modelo não possuía esse novo campo. Para adicionar, é necessário fazer uma migração. No terminal rode o seguinte comando:

```
    npx sequelize-cli migration:generate --name add-post-belongs-
user
```

Esse comando irá gerar um arquivo de migração que precisa ser implementado e ao final deve ficar conforme o modelo abaixo:

```
    'use strict';

    module.exports = {
        up: async (queryInterface, Sequelize) => {
            return queryInterface.addColumn('Posts', 'userId', {
                type: Sequelize.INTEGER,
                references: {
                    model: 'Usuarios',
                    key: 'id'
                },
                onDelete: 'SET NULL'
            })
        },

        down: async (queryInterface, Sequelize) => {
            return queryInterface.removeColumn('Posts', 'userId')
        }
    };
```

Para testar se está tudo funcionando, é preciso executar todas as migrações. Então, no terminal, digite o seguinte comando:

```
npx sequelize-cli db:migrate
```

Atualizando a validação

Agora, para salvar um post, deve ser passado também o usuário autor daquele post. Para fazer isso, é necessário implementar o arquivo *post.schema.js*.

Ao final das modificações, o arquivo deve ficar conforme o modelo abaixo:

```
    module.exports = {
            type: "object",
            properties: {
                    titulo: {type: "string", maxLength: 100,
minLength: 5},
                    texto: {type: "string"},
```

```
                userId: {type: "integer"}
            },
            required: ["titulo", "texto", "userId"],
            additionalProperties: false
    }
```

Ajustando consultas

Link do video da aula: https://youtu.be/lWWptvXqB0Q

Para finalizar a aula, serão feitos ajustes nas consultas realizadas ao banco de dados. Algumas informações retornadas ao usuário significam pouco ou são irrelevantes. Para aprimorar essas consultas, acompanhe a aula e altere o arquivo de post.rota.js para melhoria nas informações exibidas ao usuário. Ao final, seu código deve ficar de acordo com o código abaixo:

```
    const express = require('express')
    const router = express.Router()
    const postMid = require('../middleware/validarPost.middleware')
    const { Post, Usuario } = require('../db/models')

    router.post('/', postMid)
    router.put('/', postMid)

    router.get('/', async (req, res) => {
            const posts = await Post.findAll()
            res.json({posts: posts})
    })

    router.get('/:id', async (req, res) => {
            const post = await Post.findByPk(req.params.id,
                    {include: [{model: Usuario}], raw: true, nest:
true})

            const postProcessado = prepararResultado(post)

            res.json({posts: postProcessado})
    })

    router.post('/', async (req, res) => {
            const post = await Post.create(req.body)
```

```
            res.json({msg: "Post adicionado com sucesso!"})
    })

    router.delete('/', async (req, res) => {
            const id = req.query.id
            const post = await Post.findByPk(id)
            if (post){
                    await post.destroy()
                    res.json({msg: "Post deletado com sucesso!"})
            }else{
                    res.status(400).json({msg: "Post não
encontrado!"})
            }
    })

    router.put('/', async (req, res) => {

            const id = req.query.id
            const post = await Post.findByPk(id)

            if (post){
                    post.titulo = req.body.titulo
                    post.texto = req.body.texto
                    await post.save()
                    res.json({msg: "Post atualizado com sucesso!"})
            }else{
                    res.status(400).json({msg: "Post não
encontrado!"})
            }

    })

    function prepararResultado(post){
            const result = Object.assign({}, post)

            if (result.createdAt) delete result.createdAt
            if (result.updatedAt) delete result.updatedAt
            if (result.userId) delete result.userId
            if (result.Usuario){
                    if (result.Usuario.senha) delete
result.Usuario.senha
            }
            return result
    }
```

```
    module.exports = router
```

Resumo

Nesta aula, você viu como organizar os arquivos de códigos da aplicação, criar semestes utilizando a ferramenta *seeders* disponível no *sequelize* e, após isso, viu como criar relações entre modelos no banco de dados. Agora, você está pronto para seguir rumo à próxima aula, na qual será ministrado o conteúdo de *Upload de Arquivos*.