CAPÍTULO 5 – Acesso e manipulação de dados com Express e MySQL

Neste capítulo você vai ver:

- Criação de banco de dados e tabelas utilizando o MySQL;
- Desenvolvimento de API-Rest com acesso ao banco de dados usando Express;
- Rotas HTTP dos tipos GET, POST, UPDATE e DELETE;
- Operações CRUD utilizando a linguagem SQL;
- Utilização do software Postman para testar a API;

Atividade 1 – Criação de banco de dados e tabela

Essa atividade tem como objetivo:

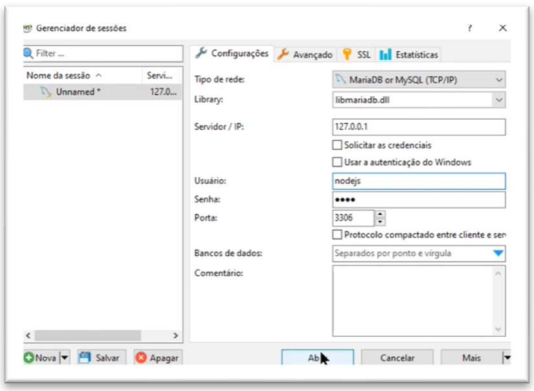
- Criar um banco de dados e uma tabela no MySQL;

Exemplo de criação de tabelas no MySQL

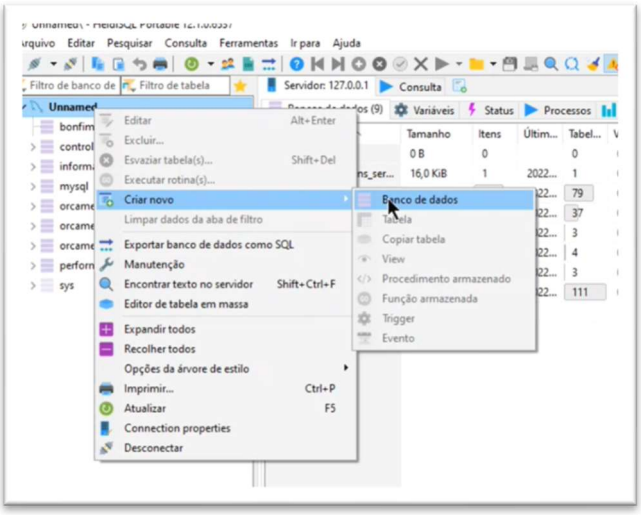
Para esta atividade utilizaremos o SGBD MySQL e um aplicativo cliente chamado HeidiSQL para realizar as configurações, detalhes de instalação destes softwares podem ser vistos no apêndice deste material. Vale ressaltar que outros softwares podem ser utilizados para o mesmo fim tais como Xampp e Mysql Workbench.

Para iniciar a atividade vamos começar do pressuposto de que o MySQL e o HeidiSQL estão devidamente instalados e configurados.

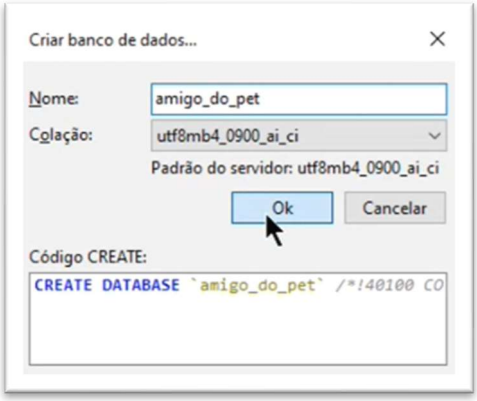
1. Abra o HeidiSQL e realize o login utilizando o usuário e senha cadastrado no PC local.



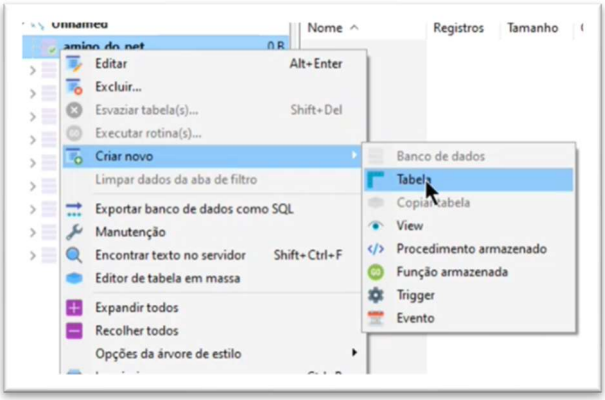
2. Na próxima tela clique com o botão direito na raiz da estrutura hierárquica do MySQL e acesse o menu “Criar novo” e depois “Banco de dados”. Conforme pode ser observado na imagem abaixo:



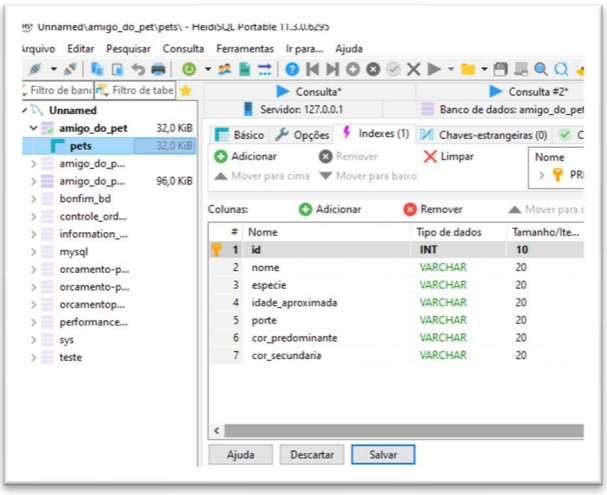
3. Crie o banco com o nome “amigo_do_pet” para utilizarmos como exemplo.



4. Agora clique com o botão direito no banco de dados criado acesse o menu “Criar novo” e depois “Tabela” e defina o nome da tabela como “pets”.



5. Adicione os seguintes atributos com as respectivas configurações e clique em salvar:



Atividade 2 – Acesso e transação com o banco de dados

Essa atividade tem como objetivo:

- Desenvolver o script de conexão com o banco de dados;

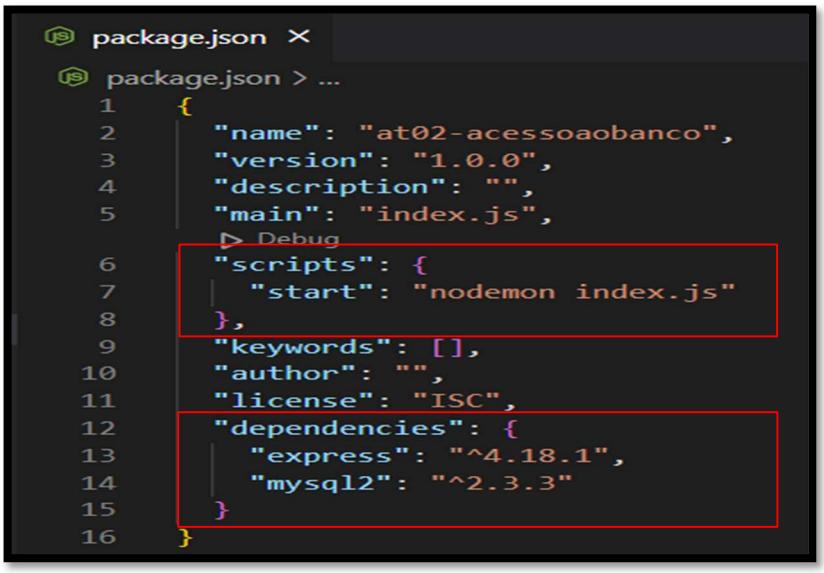
- Desenvolver uma rota GET e uma rota POST que transacionam com o banco de dados,

Comandos: npm init -y | npm i express mysql2 | mysql.createConnection

Parâmetros na url da requisição

Vamos ver como criar uma conexão com o banco de dados e realizar leitura e gravação de dados

1. Para iniciar a atividade, crie um diretório Cap05, dentro dele um subdiretório com o nome “at02-AcessoAoBanco” e depois acesse-o via console;
2. Vamos agora dar o comando para fazer inicialização de um projeto node.js. Acessando o diretório “at02-AcessoAoBanco” emita o comando “**npm init -y**” e aguarde os arquivos “package.json” ser criado.
3. Depois vamos fazer a instalação dos módulos necessários para o nosso projeto. Emita o comando “**npm i express mysql2**”.
4. No arquivo “package.json” insira na chave o script para inicialização via nodemon inserindo a **chave/valor** “start”:“nodemon index.js”, isso permitirá iniciar a aplicação apenas com o comando “**npm start**”. O conteúdo do package.json deve ficar similar ao da imagem a seguir, note também a lista de dependências:



5. Após concluir a inicialização do projeto, podemos reaproveitar os templates HTMLs do último exercício do capítulo 4. Por isso vá até a pasta do exercício “Cap04\at05-NeDB” copie a pasta “templates” e cole na pasta do nosso projeto “at02-AcessoAoBanco”.

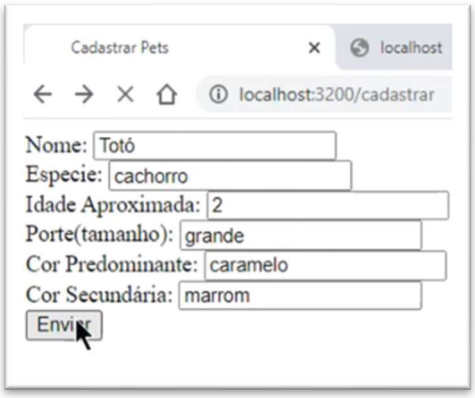
6. O próximo passo será criar o módulo de conexão com o banco, para isso crie um arquivo no diretório do projeto com o nome “bd.js” e insira o código para a conexão. Para o nosso exemplo ficará com o seguinte código:

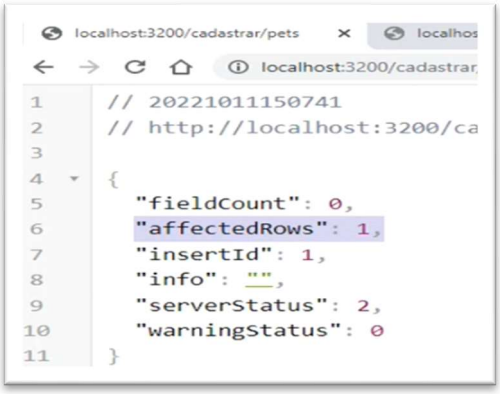
```
const mysql = require('mysql2') //Importação do driver do MySQL
//Conexão com o banco de dados
module.exports = ()=> {
    const connection = mysql.createConnection({
        host: 'localhost', //Nome DNS ou IP da hospedagem do banco
        user: 'nodejs', //Usuário do MySQL
        password: '1234', //Senha do MySQL
        database: 'amigo_do_pet' //Nome do Banco da aplicação
    });
    return connection
}
```

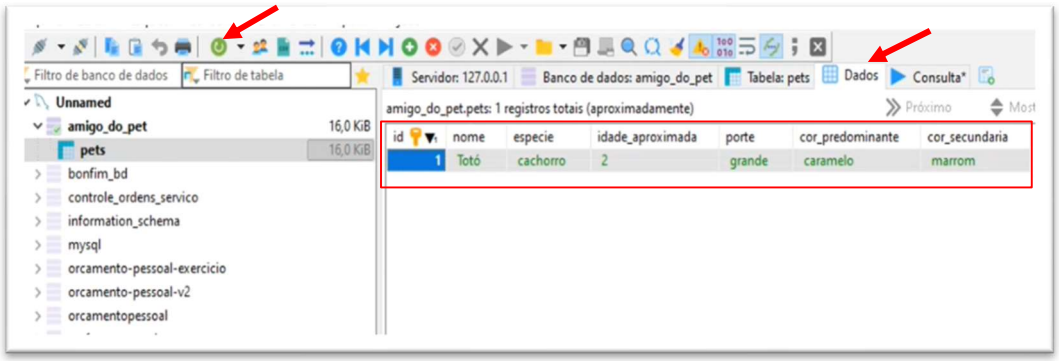
7. Agora implemente o script principal criando o arquivo index.js com o seguinte código:

```
const express = require('express')
const app = express()
const con = require('./bd')();
const path = require('path')
const baseDir=path.join(__dirname, 'templates')
var porta = '3200'
app.use(express.urlencoded({extended:true}))
app.use(express.json())

app.get('/', (req, res)=>res.sendFile(`${baseDir}/index.html`))
app.get('/cadastrar', (req, res)=>res.sendFile(`${baseDir}/cadastrar.html`))

app.post('/cadastrar', async (req, res)=>{
    let {nome, especie, idadeAproximada, porteTamanho, corPredominante,
corSecundaria} = req.body
    let dados = [nome, especie, parseFloat(idadeAproximada), porteTamanho,
corPredominante, corSecundaria]
    const sql = "INSERT INTO pets (nome, especie, idade_aproximada, porte,
cor_predominante, cor_secundaria) VALUES (?,?,?, ?, ?, ?);";
    try {
       con.query(sql, dados, (erro, resp)=>{
            let resposta
            if(erro) resposta = {...erro, status:400, message: `Os dados não
foram gravados`}
            else resposta = {...resp, status:200, message: `Sucesso:
${dados.affectedRows} linha(s) alterada(s)`}
            res.json(resposta).status(200)
        })
    } catch (erro) {
        res.send('Erro ao acessar o banco: '+erro).status(400);
    }
})
app.get('/registros', (req, res)=>{
    //ler dados
    const sql = 'SELECT * FROM pets';
    try {
        con.query(sql, (erro, dados)=>{
            if(erro) resposta = {...erro, status:400, message: `Os dados não
foram gravados`}
            else resposta = {...dados, status:200, message: `Sucesso!`}
            res.json(resposta).status(200)
        });
    } catch (erro) {
        res.send('Erro ao acessar o banco: '+erro).status(400);
    }
})
app.listen(porta, ()=>console.log(`Servidor rodando em:
http://localhost:${porta}`))
```

8. Por fim, podemos iniciar o serviço da API com o comando “npm start” e realizar os testes de inserção de registros e leitura dos dados no banco através das rotas POST e GET que foram criadas.







Atividade 3 – API-Rest com CRUD completo

Essa atividade tem como objetivo:

- Desenvolver uma API Rest com rotas utilizando os verbos HTTP - GET, POST, PUT e DELETE capaz de executar as transações CRUD o MySQL.

Comandos: mysql2/promise, INSERT, SELECT, UPDATE e DELETE, DESCRIBE

CRUD com API-Rest(verbos HTTP – GET, POST, PUT e DELETE)

Vamos desenvolver um código otimizado de uma API capaz de realizar as transações de inserção, leitura, atualização e exclusão de registros no MySQL através de uma aplicação Express que implementa as principais rotas HTTP (GET, POST, PUT e DELETE). Trabalharemos com o as funcionalidades assíncronas de conexão e acesso ao banco disponíveis no modulo mysql2/promise.
Nesta atividade teremos **dois módulos**, o modulo **bd.js** contendo o script de conexão com o banco e métodos de transação(CRUD) com o banco. Teremos também o modulo **index.js** contendo a aplicação Express e as rotas que consomem os métodos exportados no bd.js.

1. Para iniciar a atividade, crie um subdiretório dentro de Cap05 com o nome **at03-CrudCompleto** e depois acesse-o via console;

2. Inicialize o projeto, instale os módulos **express** e **mysql2** e faça as alterações no arquivo package.json para executar com o nodemon conforme os passos 2,3 e 4 da atividade anterior.
3. Crie um novo arquivo no subdiretório at03-CrudCompleto com o nome **bd.js**. Aqui implementaremos o script de conexão e os métodos do CRUD, todos de forma assíncrona.
4. No bd.js insira o código que trata da conexão com o banco, implemente o seguinte código:

```
//Objeto de conexão
const chaveAcesso = {
    host: 'localhost',
    user: 'nodejs',
    password: '1234',
    database: 'amigo_do_pet'
}
//Função de conexão
const conectar = async () => {
    if (global.statusConexao){
        return global.conexao
    }
    const mysql = require('mysql2/promise')
    const con = await mysql.createConnection(chaveAcesso)
    global.conexao = con
    global.statusConexao=true
    console.log('Conectado ao Banco: ', global.statusConexao)
    return con
}
```

5. Agora vamos implementar de uma função que é capaz de executar qualquer transação com o banco com base em instruções SQL e retorne uma resposta de sucesso ou erro. Segue o código:

```
//Função que recebe a string SQL e executa a transação no banco
const executarQuery = async (sql, dados='') => {
    let con = await conectar()
    try {
        let respBd = await con.query(sql, dados)
        respBd= respBd[1]? respBd[0]: respBd
        return await {dados:[...respBd], status:200, message:'Sucesso'}
    } catch (erro) {
        return {status:400, message:'Inconsistência nas informações: '+erro}
    }
}
```

6. Logo abaixo implementaremos uma função capaz de consultar todos os campos de uma tabela e retorná-los em um array. Essa função será utilizada em todos os métodos do CRUD para a construção da string contendo a instrução SQL. Segue o código:

```
//Função que obtem os campos de uma tabela
const obterCampos = async(tabela)=>{
    let con = await conectar()
```

```
    try {
        let sql= `DESCRIBE ${tabela}`
        let campos = await con.query(sql)
        return await campos[0]
    } catch (erro) {
        return `${tabela} não encontrada: ${erro}`
    }
}
```

7. Para finalizar o módulo bd.js, implementaremos agora os métodos a serem exportados: inserir, ler, atualizar e deletar:

```
//Métodos do CRUD a serem exportados e consumidos pelo app principal
module.exports={
    inserir: async (tabela, dados)=>{
        let campos = await obterCampos(tabela) //Obter campos do bd
        campos = campos.map((el)=>el.Field) //Tratamento na string com campos
        campos.shift()
        let argumentos = ''
        campos.forEach(el =>argumentos+=',?')//String com sequência de '?'
        argumentos = argumentos.slice(1)
        campos = campos.toString()
        const sql=`INSERT INTO ${tabela} (${campos}) VALUES (${argumentos});`
        return await executarQuery(sql, dados) //Salvar dados
    },
    ler: async (tabela, id='') => {
        let sql = id==''? `SELECT * FROM ${tabela}`: `SELECT * FROM ${tabela}
WHERE id = ${id};`
        let linhas = await executarQuery(sql)
        linhas.dados = linhas.dados.length==0? linhas.dados=[{message:'Nenhum
registro encontrato'}]:linhas.dados
        return await linhas
    },
    atualizar: async(tabela, dados, id) => {
        let campos = await obterCampos(tabela)
        campos.shift()
        campos = campos.map((el)=>el.Field+'=?')
        campos = campos.toString()
        let sql= `UPDATE ${tabela} SET ${campos} WHERE id =?`
        dados.push(id)
        return await executarQuery(sql, dados)
    },
    deletar: async (tabela, id) => {
        let sql = `DELETE FROM ${tabela} WHERE id = ${id};`
        return await executarQuery(sql)
    }
}
```

8. Vamos criar o módulo **idex.js** no subdiretório **at03-CrudCompleto** e implementar o código instanciando a aplicação Express com os parâmetros já vistos em atividades anteriores, segue o código:

```
const express = require('express')
const app = express()
const bd = require('./bd')
var porta = '3200'
app.use(express.urlencoded({extended:true}))
app.use(express.json())
```

9. Vamos agora implementar as rotas POST, GET, PUT e DELETE que consomem os métodos do CURD no modulo bd.js. Iniciaremos com a rota POST – **"/cadastrar/:tabela"**, segue o código:

```
app.post('/cadastrar/:tabela', async (req, res)=>{
    try {
      let dados = Object.values(req.body).map((val)=>val)
      let tabela = req.params.tabela
      let respBd= await bd.inserir(tabela, dados)
      res.json(respBd).status(200)
    } catch (erro) {
      res.json(erro).status(400)
    }
})
```

10. Acrescente o código das próximas duas rotas GET: **"/consultar/:tabela"** busca todos os registros e **"/consultar/:tabela/:id"** busca apenas um registro baseado no ID. Segue o código:

```
app.get('/consultar/:tabela',async (req, res) => {
  try {
  let tabela = req.params.tabela
  let respBd = await bd.ler(tabela)
  res.json(respBd).status(200)
  } catch (erro) {
    res.json(erro).status(400)
  }
})
app.get('/consultar/:tabela/:id', async (req, res) => {
    try {
      let {tabela, id}= req.params
    let respBd = await bd.ler(tabela, id)
    res.json(respBd).status(200)
    } catch (erro) {
      res.json(erro).status(400)
    }
  })
```

11. Implemente rota PUT: **"/editar/:tabela/:id"** que atualizar os registros. Segue o código:

```
app.put('/editar/:tabela/:id', async (req, res) => {
    try {
      let {tabela, id} = req.params
      let dados = Object.values(req.body).map((val)=>val)
      let respBd = await bd.atualizar(tabela, dados, id)
      res.json(respBd).status(200)
    } catch (erro){
      res.json(erro).status(400)
    }
  })
```

12. Por último, vamos implementar a rota DELETE: "/excluir/:tabela/:id" que realiza a exclusão de registros. Segue o código:

```
  app.delete('/excluir/:tabela/:id', async (req, res) => {
    try {
      let {tabela, id} = req.params
      let respBd = await bd.deletar(tabela, id)
      res.json(respBd).status(200)
    } catch (erro) {
      res.json(erro).status(400)
    }
  })
```

13. Para finalizar o script do modulo index.js não podemos esquecer da linha que inicia o serviço da aplicação para "escutar" as requisições no protocolo HTTP:

```
app.listen(porta, ()=>console.log(`Servidor rodando em:
http://localhost:${porta}`))
```

14. Execute o script index.js através do comando **npm start**. Caso algum erro for relatado execute o processo de debug conferindo as linhas indicadas no erro ou nas funções relacionadas. Testaremos as funcionalidades de transação com o banco de dados da nossa API na próxima atividade utilizando a ferramenta de teste POSTMAN.

Atividade 4 – Testes de funcionalidades da API-Rest

Essa atividade tem como objetivo:

- Realizar um teste de todas as funcionalidades da API, enviando requisições a todas as rotas criadas – GET, POST, PUT e DELETE

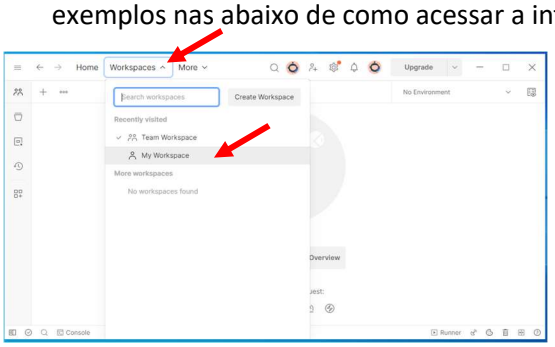
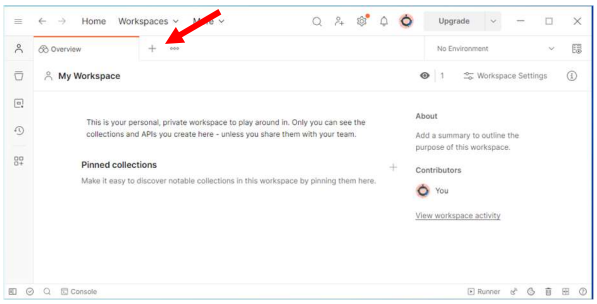
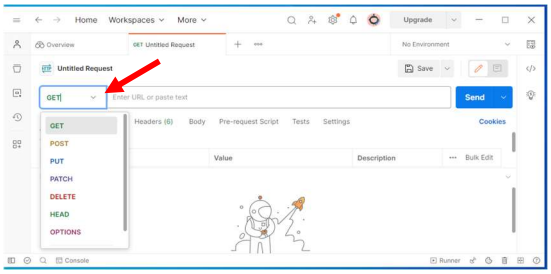
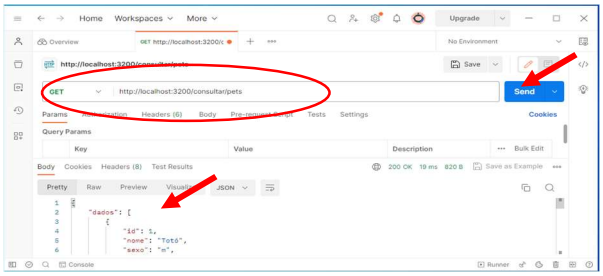
Teste funcionalidades da API utilizando o POSTMAN

Até o momento para consumir os recursos da API utilizamos formulários HTML. Porém os formulários HTML só são capazes de enviar requisições HTTP utilizando os verbos

GET e POST por isso iremos utilizar uma ferramenta chamada POSTMAN que nos permite realizar o teste completo da API.
Existe uma versão portable do Postman nos recursos deste curso, ele também pode ser baixado na no link https://www.postman.com/downloads/, pode ser utilizado online, como uma extensão do chrome ou ainda extensão do VSCODE. É necessário realizar um cadastro ou fazer login com conta google.
Esta ferramenta será utilizada também no próximo capítulo.

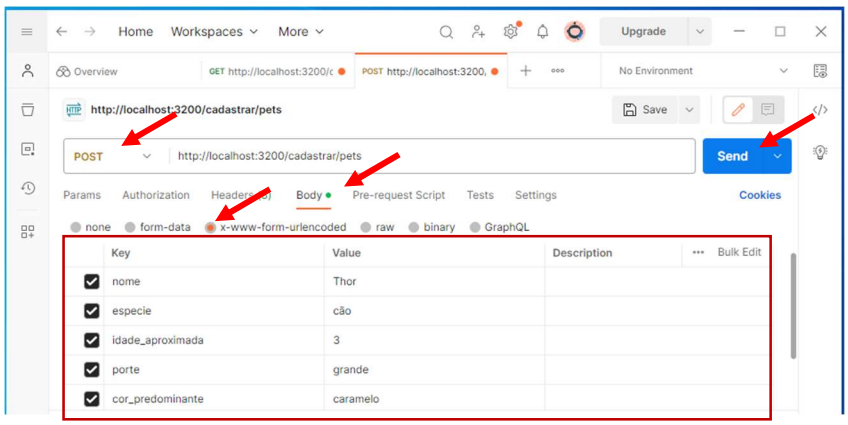
1. Inicie a atividade instalando o Postman e realizando o login, pode-se utilizar a conta Google.
2. Com a aplicação rodando, envie requisições HTTP para cada rotas da API e verifique o retorno.

   É muito importante que cada rota seja enviada para o verbo HTTP correspondente e o caminho esteja exato. Envie uma requisição GET para a rota http://localhost:3200/consultar/pets. Veja exemplos nas abaixo de como acessar a interface My Workspace e realizar requisições:

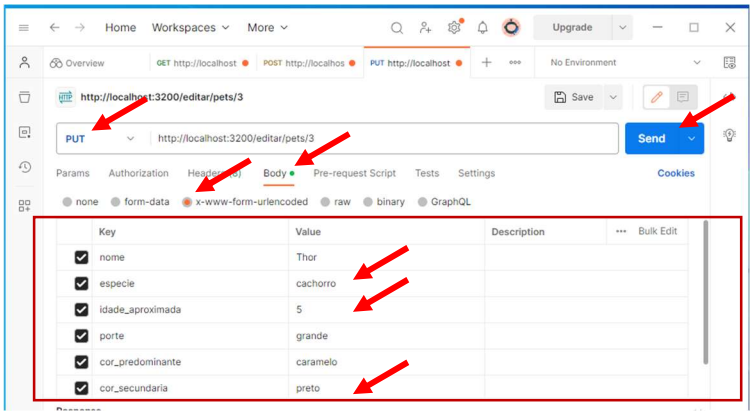





3. Crie outra aba e envie uma requisição para a rota POST http://localhost:3200/cadastrar/pets.

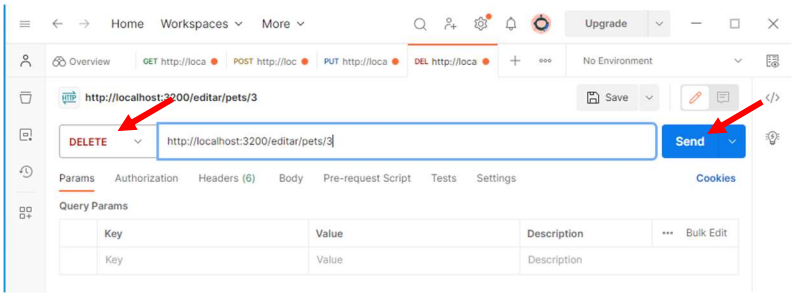
   Neste caso precisamos passar os dados no corpo da requisição, siga as indicações na imagem a seguir:



4. Crie outra aba e envie uma requisição para a rota PUT http://localhost:3200/editar/pets/3. Neste caso os dados também precisam ser passados no corpo da requisição, mas também é necessário passar o id do registro a ser editado, como no exemplo o id 3, altere alguns dados do último registro e após a transação confira as alterações no banco de dados. Para realizar um teste siga as indicações na imagem a seguir:



5. Crie outra aba e envie uma requisição para a rota DELETE http://localhost:3200/excluir/pets/3. Desta vez, nenhum dado será enviado no corpo da requisição, mas é necessário passar o id do registro a ser excluído, como no exemplo o id 3 será removido, após a transação confira no banco de dados se o registro 3 foi removido. Para realizar um teste siga as indicações na imagem a seguir:



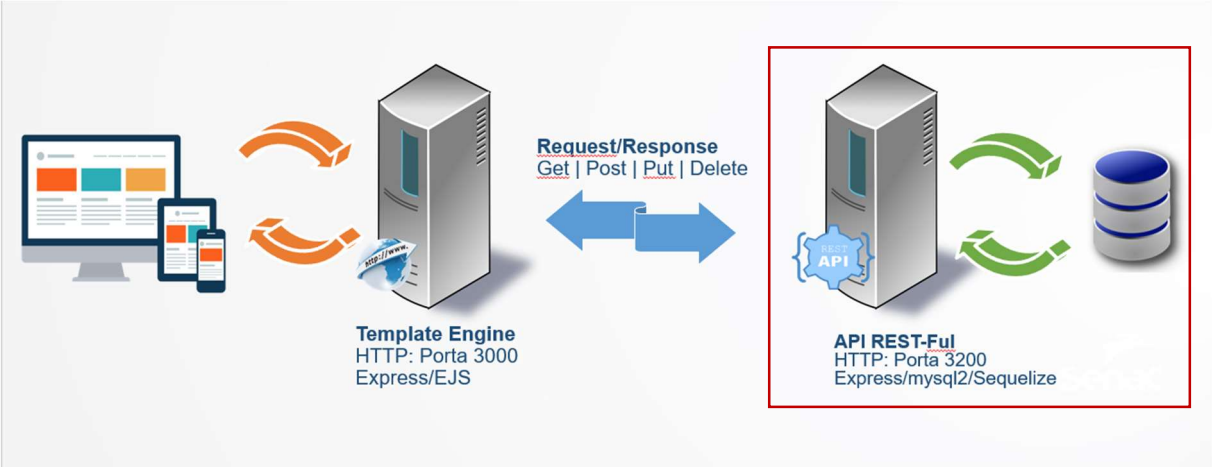
Exercícios extras

Estas atividades extras têm como objetivo criar novas tabelas de dados. Realizar testar as funcionalidades da API realizando as operações de CRUD. Exemplo:

1. Crie outra tabela no banco de dados, como por exemplo “usuários”, com campos como:
   - ✓ id
   - ✓ nome
   - ✓ cpf
   - ✓ email
   - ✓ senha
2. Realize os testes de funcionalidade utilizando o Postman nas rotas GET, POST, PUT e DELETE para a nova tabela.
3. Outras tabelas podem ser criadas a fim de repetir o procedimento de testes.

CAPÍTULO 6 – Manipulação de dados utilizando ORM

A proposta para esta aplicação é desenvolver dois servidores, conforme a imagem a seguir:



Chegou a hora de desenvolver uma versão funcional da API-Rest que fornecerá serviços para a aplicação web.

Neste capítulo você vai ver:

- Manipular tabelas do banco de dados utilizando models do ORM Sequelize;
- Criar relacionamento entre tabelas(models);
- Configuração de CORS;
- Utilização de Middleware;
- Organização do código em módulos no diretório “controllers”;
- Utilização do módulo “consign” para organizar o código.

Atividade 1 – Introdução e instalação do ORM - Sequelize

A sigla ORM significa **Object Relational Mapping**, tem como função principal aproximar o paradigma de orientação a objetos ao paradigma de **bancos de dados relacionais** (MySQL, PostigreSQL, SQL Server, Oracle e outros). Basicamente mapeia as entidades (tabelas) através de models(classes), a fim de abstrair complexidades do SGBD, tais como instruções SQL e relacionamentos entre tabelas.

Essa atividade tem como objetivo:

- Instalar o módulo Sequelize;

- Criar o módulo de conexão com o banco;

- Criar um model;

- Utilizar os principais métodos de manipulação de dados.

Comandos:

- **npm init** -y | **npm install** express mysql2 sequelize

Exemplo de aplicação CRUD utilizando Express e Sequelize

Vamos ver como criar uma conexão com o banco de dados, um model e utilizar os principais métodos do Sequelize.

1. Para iniciar a atividade, crie um diretório **Cap06**, dentro dele crie um subdiretório com o nome "**at01-Sequelize**" e abra no VSCode;
2. Realize o procedimento para inicializar o projeto e instalar os módulos necessários com os comandos no console:
   - ✓ npm init -y
   - ✓ npm install express mysql2 sequelize

   Insira também o **"start"**: "nodemon index.js" (arquivo pakete.json vide exemplos anteriores)
3. Crie no MySql um novo banco(Schema) com o título de "teste".
4. Crie um módulo chamado "bdConexao.js" e insira o seguinte código:

```
//Desestruturação da classe Sequelize
const {Sequelize} = require('sequelize')
//Instancia de um objeto Sequelize
const sequelize = new Sequelize(
    //Lista de parâmetros: banco, usuário e senha
    'teste', 'nodejs', '1234', {
        host:'localhost', //nome DNS ou IP do servidor
        dialect:'mysql', //SGBD utilizado
        charset: 'utf8', //tabela de caracteres
        collate: 'utf8_general_ci', //colação
        timezone:"-03:00" //fuso horário
    })
try {
    //metodo de autenticação que conecta ao banco
    sequelize.authenticate()
    console.log('Conectado ao banco')
} catch (erro) {
    console.log('Não foi possível conectar: ', erro )
}
module.exports = sequelize
```

5. Vamos criar o model para representar a entidade(tabela) usuário. Para isso crie um subdiretório no projeto chamado “models” que irá abrigar os models. Crie dentro de “models” um arquivo chamado “usuario.js” e insira o seguinte código:

```
//Desestruturação das classes DataTypes e Model do módulo sequelize
const { DataTypes, Model } = require('sequelize')

//Importação do módulo de conexão
const sequelize = require('../bdConexao')

//Classe Usuario herdando a classe Model
class Usuario extends Model{}
Usuario.init({
    //Atributos da entidade usuário
    nome: {
        type: DataTypes.STRING,
        allowNull: false
      },
      cpf: {
        type: DataTypes.STRING,
        allowNull:false
      }
},{
    sequelize,
    modelName:'usuario'
  }
)

sequelize.sync() //Cria a tabela caso não exista
 module.exports = Usuario

 /*
   Consulte todos os tipos de dados da classe dataTypes em:
   https://sequelize.org/docs/v6/core-concepts/model-basics/#data-types
  */
```

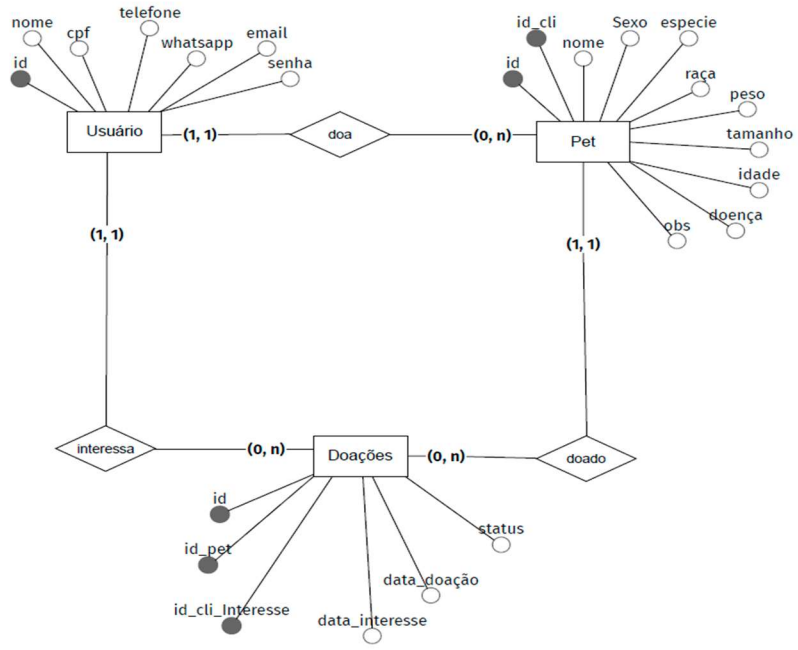
6. Saia do diretório “models” e no diretório raiz do projeto crie o módulo index.js e insira o seguinte código:

```
const express = require('express')
const app = express()

//Instância do objeto (Model)
const usuario = new require('./models/usuario')
var porta = '3200'

app.use(express.urlencoded({extended:false}))
app.use(express.json())

app.get('/', (req, res)=>res.send('API - Amigo do Pet'))

app.get('/consultar/usuarios/:id?', async (req, res)=>{
    //O método finOne busca um registro baseado em uma condição, por
exemplo o id
    // O método findAll obtém todos os registros da entidade
    let dados = req.params.id? await
usuario.findOne({where:{id:req.params.id}}) :
    await usuario.findAll()
    res.json(dados)
})
app.post('/cadastar/usuarios', async (req, res)=>{
    let dados = req.body
    //Método create insele novo registro
    let respBd = await usuario.create(dados)
    res.json(respBd)
})
app.put('/atualizar/usuarios/:id', async (req, res) => {
    let id = req.params.id
    let dados = req.body
    //Método update atualiza o registro baseado no id
    let respBd = await usuario.update(dados, {where:{id:id}})
    res.json(respBd)
})
app.delete('/excluir/usuarios/:id', async (req, res) => {
    let id = req.params.id
    //Método destry exclui um registro baseado no id
    let respBd = await usuario.destroy({where:{id:id}})
    res.json(respBd)
})
app.listen(porta, ()=>console.log(`Servidor rodando em:
http://localhost:${porta}`))
```

7. Rode o código com o comando **npm start**.
8. Execute o procedimento de testes nas rotas GET, POST, PUT e DELETE utilizando o Postmann conforme atividade 4 do capítulo 5.

Atividade 2 – Desenvolvimento de API – REST

Para esta atividade vamos considerar as entidades – Usuário – Pet – Doações os atributos e relacionamentos definidos no MER a seguir:



Essa atividade tem como objetivo:

- Criar models com definição de relacionamentos entre as entidades;

- Estruturar o código da aplicação em módulos no diretório controllers.

Comandos:

- **npm init** -y | **npm install** express mysql2 sequelize cors

Desenvolvimento da versão 1.0 da API do sistema "Amigo do Pet".

Nesta aplicação vamos definir as tabelas, campos e relacionamentos do banco de dados através de models do Sequelize. E estruturar o código da aplicação em módulos, utilizando a biblioteca "consign" e o diretório controllers.

1. Para iniciar a atividade, crie no diretório **Cap06** um subdiretório com o nome "**at02-APIRest_Amigo_do_Pet-V_1_0**" e abra no VSCode;
2. Realize o procedimento para inicializar o projeto e instalar os módulos necessários com os comandos no console:
   - ✓ npm init -y
   - ✓ npm install express mysql2 sequelize cors

Insira também o **"start"**: "nodemon index.js" (arquivo pakete.json vide exemplos anteriores)

3. Crie no MySql um novo banco(**Schema**) com o título de "**amigo_do_pet**" - Caso o banco de dados dos capítulos anteriores ainda existir, deve ser apagado e criado novamente.
4. Crie um módulo chamado "**bdConexao.js**" e insira o seguinte código:

```
const {Sequelize} = require('sequelize');
const sequelize = new Sequelize(
    'amigo_do_pet', 'nodejs', '1234', {
        host:'localhost',
        dialect:'mysql',
        charset: 'utf8',
        collate: 'utf8_general_ci',
        timezone:"-03:00"
    })
try {
    sequelize.authenticate()
    console.log('Conectado ao banco')
} catch (erro) {
    console.log('Não foi possível conectar: ', erro )
}
module.exports = sequelize
```

5. Crie no diretório raiz do projeto o módulo index.js e insira o seguinte código:

```
const express = require('express')
//Módulo consign facilita a organização de módulos e faz carga automático
de scripts
const consign=require('consign')
const app = express()
const cors = require('cors')
var porta = '3200'
app.use(cors()) //Configura política de segurança CORS
app.use(express.urlencoded({extended:false}))
app.use(express.json())
app.get('/', (req, res)=>res.send('API - Amigo do Pet'))

consign() //Execução do consign
    .include('./controllers/rotas') //Inclui  módulos contidos no diretório
especificado
    .into(app) //passa a instância do Express o para os módulos

app.listen(porta, ()=>console.log(`Servidor rodando em:
http://localhost:${porta}`))
```

6. Crie um subdiretório no projeto chamado "models" para os models. Crie dentro de "**models**" um arquivo chamado - "**usuario.js**" e insira o seguinte código:

```
const { DataTypes, Model } = require('sequelize')
const sequelize = require('../bdConexao')
class Usuario extends Model{}
Usuario.init({
    nome: {
        type: DataTypes.STRING(50),
        allowNull: false
      },
      cpf: {
        type: DataTypes.STRING(14),
        allowNull:false
      },
      telefone: {
        type: DataTypes.STRING(14),
        allowNull:true
      },
      whatsapp: {
        type: DataTypes.STRING(14),
        allowNull:true
      },
      email: {
        type: DataTypes.STRING(50),
        allowNull:false
      },
      senha: {
        type: DataTypes.STRING,
        allowNull:false
      }
},{
    sequelize,
    modelName:'usuario'
  }
)
sequelize.sync()
 module.exports = Usuario
```

7. Crie dentro de “**models**” outro arquivo chamado - “**pet.js**” e insira o seguinte código:

```
const { DataTypes, Model } = require('sequelize')
const sequelize = require('../bdConexao')
const usuario = new require('./usuario')
class Pet extends Model{}
Pet.init({
    nome: {
        type: DataTypes.STRING(50),
        allowNull: false
      },
      sexo: {
        type: DataTypes.STRING(1),
        allowNull:false
      },
      especie: {
        type: DataTypes.STRING(50),
        allowNull:false
      },
      raca: {
        type: DataTypes.STRING(50),
        allowNull:true
      },
      peso: {
        type: DataTypes.INTEGER,
        allowNull:true
      },
      tamanho: {
        type: DataTypes.STRING(10),
        allowNull:false
      },
      idade: {
        type: DataTypes.INTEGER,
        allowNull:true
      },
      doenca: {
        type: DataTypes.STRING,
        allowNull:true
      },
      obs: {
        type: DataTypes.STRING,
        allowNull:true
      }
},{
    sequelize,
    modelName:'pet'
  })
usuario.hasMany(Pet) // Usuario tem muitos Pets 1-p-M
Pet.belongsTo(usuario) //Pet pertence a Usuário 1-p-1
sequelize.sync()
 module.exports = Pet
```

8. Crie dentro de “**models**” um arquivo chamado - “**doacao.js**” e insira o seguinte código:

```
const { Sequelize, DataTypes, Model } = require('sequelize')
const sequelize = require('../bdConexao')
const usuario = new require('./usuario')
const pet = new require('./pet')
class Doacao extends Model{}
Doacao.init({
    data_interesse: {
        type:Sequelize.DATEONLY,
        defaultValue: DataTypes.NOW
      },
      data_doacao:{
        type:Sequelize.DATEONLY,
        allowNull: true
      },
      status: {
        type: DataTypes.STRING,
        defaultValue:"Cadastrada"
      }
    },{
      sequelize,
      modelName:'doacoes'
    }
  )
  pet.hasMany(Doacao) //Muitos Pets têm muitas Doações - M-p-M
  Doacao.belongsTo(pet)

  usuario.hasMany(Doacao) //Muitos Usuários têm muitas Doações - M-p-M
  Doacao.belongsTo(usuario)

  sequelize.sync()
   module.exports = Doacao
```

9. Crie um subdiretório no projeto chamado “**controllers”** e dentro dele outro subdiretório chamado “**rotas**”, conforme definido no método include do consign.
10. Dentro do subdiretório “rotas” vamos criar um módulo para implementarmos as rotas de cada entidade. Vamos começar criando em “rotas” um arquivo “**usuario.js**” e inserindo o seguinte código:

```
const model = new require('../../models/usuario')
const rota = 'usuarios'
module.exports = (app)=>{
    app.get(`/consultar/${rota}/:id?`, async (req, res)=>{
        try {
            let dados=req.params.id?
                  await model.findOne({where:{id:req.params.id}}):
                  await model.findAll()
            res.json(dados).status(200)
        } catch (error) {
            res.json(error).status(400)
        }
    })
    app.post(`/cadastrar/${rota}`, async (req, res)=>{
        try {
            let dados = req.body
            const salt = bcrypt.genSaltSync()
            dados.senha = bcrypt.hashSync(dados.senha, salt)
            let respBd = await model.create(dados)
            res.json(respBd).status(200)
        } catch (error) {
            res.json(error).status(400)
        }
    })
    app.put(`/atualizar/${rota}/:id`, async (req, res) => {
        try {
            let id = req.params.id
            let dados = req.body
            console.log(dados)
            let respBd = await model.update(dados, {where:{id:id}})
            res.json(respBd)
        } catch (error) {
            res.json(error).status(400)
        }
    })
    app.delete(`/excluir/${rota}/:id`, async (req, res) => {
        try {
            let id = req.params.id
            let respBd = await model.destroy({where:{id:id}})
            res.json(respBd)
        } catch (error) {
            res.json(error).status(400)
        }
    })
}
```

11. Agora crie em “rotas” um módulo chamado “**pet.js**” e insira o seguinte código: