Princípios de Desenvolvimento WEB - UFCG 2024.2

LABORATÓRIO PRÁTICO
CONEXÃO BANCO DE DADOS - MySQL OU PostgreSQL

De acordo com o que aprendemos na disciplina, a estrutura do seu projeto deve estar parecida com a seguinte:

Estrutura do projeto

```
/my-app
├── /src
│   ├── /config          # Config Banco de Dados
│   │   ├── database.ts
│   ├── /repository      # Camada de repositório
│   │   ├── userRepository.ts
│   ├── /services        # Camada de serviço
│   ├── /controllers     # Camada de controllers
│   ├── /models          # Definição de modelos
│   ├── index.ts         # Arquivo de início
├── package.json
├── tsconfig.json
├── .env
```

1. **Instalação das dependências**

   Primeiro, instale as bibliotecas necessárias:

   1. npm init -y
   2. npm install express dotenv pg mysql2 sequelize
   3. npm install --save-dev typescript ts-node @types/node @types/express @types/dotenv

   ➢ Explicação:
      i. **pg** → Para conexão com **PostgreSQL**
      ii. **mysql2** → Para conexão com **MySQL**
      iii. **sequelize** → ORM para facilitar a manipulação do banco
      iv. **dotenv** → Para gerenciar variáveis de ambiente
      v. **express** → Para criar uma API

2. Configuração de Conexão com Banco de Dados

No arquivo .env, adicione as credenciais do banco:

```
DB_DIALECT=postgres  # ou mysql
DB_HOST=localhost
DB_PORT=5432        # 3306 para MySQL
DB_USER=seu_usuario
DB_PASS=sua_senha
DB_NAME=seu_banco
```

Agora, crie database.ts dentro da pasta config:

```
import { Sequelize } from 'sequelize'
import * as dotenv from 'dotenv'

dotenv.config();

const sequelize = new Sequelize({
  dialect: process.env.DB_DIALECT as "postgres" | "mysql",
  host: process.env.DB_HOST,
  port: Number(process.env.DB_PORT),
  username: process.env.DB_USER,
  password: process.env.DB_PASS,
  database: process.env.DB_NAME,
  logging: false,
});

export default sequelize;
```

3. Criando um Modelo de Usuário

Dentro da pasta models, crie User.ts:

```
import { Model, DataTypes, Optional } from 'sequelize';
import sequelize from '../config/database';

// Defina os atributos do modelo
interface UserAttributes {
  id: number;
  name: string;
  email: string;
```

```
  password: string;
}

export class User extends Model<UserAttributes, UserCreationAttributes> implements
UserAttributes {
  public id!: number;
  public name!: string;
  public email!: string;
  public password!: string;
}

// Inicialize o modelo com os campos no banco
User.init(
  {
    id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
    },
    name: {
      type: DataTypes.STRING,
      allowNull: false,
    },
    email: {
      type: DataTypes.STRING,
      allowNull: false,
      unique: true,
    },
    password: {
      type: DataTypes.STRING,
      allowNull: false,
    },
  },
  {
    sequelize,
    tableName: "users",
    timestamps: false,
  }
);
```

4. **Camada Repository**

Agora, crie userRepository.ts dentro de repository:

```
import { User } from "../models/User";

export class UserRepository {
   // Criar um novo usuário
  async createUser(name: string, email: string, password: string) {
    // Use o método `create` para salvar no banco de dados
    return await User.create({
      name,
      email,
      password
    });
  }

  async getAllUsers() {
    return await User.findAll();
  }
}
```

5. **Criando um servidor Express para testar**

Agora, crie um arquivo index.ts na raiz do projeto:

```
import * as express from "express";
import * as dotenv from "dotenv";
import  sequelize from "./config/database";
import { UserRepository } from "./repository/userRespository";

dotenv.config();

const app = express();
app.use(express.json());

const userRepo = new UserRepository();

app.post("/users", async (req, res) => {
  try {
    const { name, email, password } = req.body;
    const user = await userRepo.createUser(name, email, password);
    res.json(user); // Retorna o usuário criado
  } catch (error: any) {
```

```
    res.status(500).json({ message: "Erro ao criar o usuário", error: error.message });
  }
});

app.get("/users", async (req, res) => {
  try {
    const users = await userRepo.getAllUsers();
    res.json(users); // Retorna todos os usuários
  } catch (error: any) {
    res.status(500).json({ message: "Erro ao obter os usuários", error: error.message });
  }
});

// Testando a conexão e inicializando o servidor
sequelize.sync({ force: true }).then(() => {
  console.log("Banco de dados conectado!");
  app.listen(3000, () => console.log("Servidor rodando na porta 3000"));
}).catch((error) => {
  console.error("Erro ao conectar ao banco de dados:", error);
});
```

6. **Editando o arquivo tsconfig.json**

   Dentro do arquivo, coloque a seguinte configuração:

```
{
    "compilerOptions": {
      "target": "es6",
      "module": "commonjs",
      "declaration": true,
      "outDir": "./lib",
      "strict": true
    },
    "include": ["src"],
    "exclude": ["node_modules", "**/__tests__/*"]
}
```

7. **Rodando o projeto**

   Antes de rodar, crie um script no package.json:

```
"scripts": {
   "dev": "ts-node src/index.ts"
 },
```

   Agora, execute:

   ➢ npm run dev

EXTRA

O Docker é uma ferramenta incrível para criar, distribuir e executar aplicações em containers. Usá-lo para evitar baixar todas as dependências diretamente no seu computador é uma excelente maneira de garantir que sua aplicação seja executada em qualquer ambiente sem precisar se preocupar com versões de dependências, configurações locais e muito mais.

Vamos ver como você pode usar o **Docker** para evitar baixar todas as dependências diretamente no seu computador e ainda garantir que a sua aplicação seja executada corretamente em qualquer lugar.

O que é o Docker?

O Docker cria **containers** para suas aplicações. Um container é um pacote que contém a aplicação e todas as suas dependências (bibliotecas, arquivos de configuração, etc.). Isso garante que o ambiente de execução da sua aplicação seja o mesmo, independentemente do sistema operacional ou das configurações locais.

Por que usar Docker para evitar dependências no seu computador?

1. **Isolamento de Dependências**: O Docker isola a aplicação e suas dependências do sistema operacional principal, ou seja, as dependências de sua aplicação não precisam estar instaladas no seu computador.
2. **Portabilidade**: Uma vez que o Docker cria containers com tudo o que sua aplicação precisa para rodar, você pode executar a aplicação em qualquer lugar (servidores, máquinas locais, ambientes de desenvolvimento) sem se preocupar com diferenças de ambiente.
3. **Facilidade de Configuração**: Usando Docker, você pode configurar sua aplicação e suas dependências com um simples arquivo de configuração (Dockerfile e docker-compose.yml), evitando assim ter que configurar o ambiente manualmente em cada máquina.