Laboratório de Autenticação e Autorização

Pré-Requisitos

- Já existe uma tabela de usuários no MySQL.
- A conexão com o MySQL já está configurada (conforme mencionado).
- As rotas do Express já estão implementadas, mas não há autenticação ou autorização.

Objetivo:

Adicionar autenticação (login e geração de JWT) e autorização (verificação de roles) ao projeto já existente.

Passo 1: Instalar Dependências Necessárias

No seu projeto existente, instale as dependências necessárias para autenticação e autorização:

```
npm install bcryptjs jsonwebtoken dotenv
npm install --save-dev @types/bcryptjs @types/jsonwebtoken
```

- **bcryptjs**: Para criptografar e comparar senhas.
- **jsonwebtoken**: Para gerar e validar tokens JWT.
- **dotenv**: Para gerenciar variáveis de ambiente (caso ainda não esteja usando).

Passo 2: Configurar Variáveis de Ambiente

1. Adicione a chave secreta do JWT no arquivo .env (se já não existir):

```
JWT_SECRET=your_jwt_secret_key
```

2. Certifique-se de que o arquivo .env esteja sendo carregado no seu projeto. No arquivo principal (por exemplo, src/index.ts), adicione:

```
import dotenv from 'dotenv';
dotenv.config();
```

Passo 3: Criar Funções de Autenticação

1. Crie um arquivo *src/utils/auth.ts* para funções relacionadas à autenticação:

```
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';
import dotenv from 'dotenv';

dotenv.config();

const JWT_SECRET = process.env.JWT_SECRET || 'your_jwt_secret_key';

// Função para criptografar a senha

export const hashPassword = async (password: string): Promise<string> => {
  const salt = await bcrypt.genSalt(10);
  return await bcrypt.hash(password, salt);
};

// Função para comparar a senha

export const comparePassword = async (password: string, hashedPassword:
string): Promise<boolean> => {
  return await bcrypt.compare(password, hashedPassword);
};

// Função para gerar um token JWT

export const generateToken = (userId: number, username: string): string => {
  return jwt.sign({ id: userId, username }, JWT_SECRET, { expiresIn: '1h' });
};

// Função para verificar um token JWT

export const verifyToken = (token: string): any => {
  return jwt.verify(token, JWT_SECRET);
};
```

Passo 4: Criar o Controller de Autenticação

1. Crie um arquivo src/controllers/authController.ts para lidar com login e outras operações de autenticação:

```
import { Request, Response } from 'express';
import { comparePassword, generateToken } from '../utils/auth';
import { findUserByUsername } from '../models/User'; // Supondo que você já tem essa função

export const login = async (req: Request, res: Response) => {
  const { username, password } = req.body;

  try {

    // Verifica se o usuário existe

    const user = await findUserByUsername(username);
    if (!user) {
      return res.status(400).json({ message: 'Invalid username or password' });
    }

    // Compara a senha fornecida com a senha armazenada

    const isPasswordValid = await comparePassword(password, user.password);
    if (!isPasswordValid) {
      return res.status(400).json({ message: 'Invalid username or password' });
    }

    // Gera um token JWT

    const token = generateToken(user.id, user.username);

    res.status(200).json({ message: 'Login successful', token });
  } catch (err) {
    res.status(500).json({ message: 'Error logging in', error: err });
  }
};
```

Passo 5: Criar Middleware de Autenticação

1. Crie um arquivo src/middlewares/authMiddleware.ts para proteger rotas:

```
import { Request, Response, NextFunction } from 'express';
import { verifyToken } from '../utils/auth';

export const authenticate = (req: Request, res: Response, next: NextFunction) => {
  const token = req.header('Authorization')?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ message: 'Access denied. No token provided.' });
  }

  try {
    const decoded = verifyToken(token);
    (req as any).user = decoded; // Adiciona o usuário decodificado ao objeto `req`
    next();
  } catch (err) {
    res.status(400).json({ message: 'Invalid token.' });
  }
};
```

Passo 6: Adicionar Rotas de Autenticação

1. **Crie um arquivo src/routes/authRoutes.ts para as rotas de autenticação:**

```
import express from 'express';
import { login } from '../controllers/authController';

const router = express.Router();

router.post('/login', login);

export default router;
```

2. **No arquivo principal (por exemplo, src/index.ts), adicione as rotas de autenticação:**

```
import authRoutes from './routes/authRoutes';

app.use('/auth', authRoutes);
```

Passo 7: Proteger Rotas Existentes

1. Use o middleware authenticate para proteger rotas existentes. Por exemplo:

```
import { authenticate } from './middlewares/authMiddleware';

app.get('/protected', authenticate, (req, res) => {
  res.status(200).json({ message: 'You have access to this protected route' });
});
```

2. Para acessar rotas protegidas, o cliente deve enviar o token JWT no cabeçalho Authorization:

   Authorization: Bearer <token_jwt>

Passo 8: Testar a Aplicação

1. Fazer login:
   - Rota: POST /auth/login
   - Body:

```
{
  "username": "existing_user",
  "password": "existing_password"
}
```

   - Resposta: Um token JWT será retornado.

2. Acessar rota protegida:
   - Rota: GET /protected
   - Header: Authorization: Bearer <token_jwt>

Passo 9: (Opcional) Adicionar Logout

Para implementar logout, você pode simplesmente invalidar o token no lado do cliente (por exemplo, removendo o token do localStorage no frontend). No backend, você pode manter uma lista de tokens inválidos (blacklist) se necessário.