Documentação Técnica do Backend - Projeto Taskea

Introdução

Esta seção detalha a arquitetura e o funcionamento interno do backend da aplicação Taskea. O backend é construído utilizando Node.js e o framework Express, responsável por gerenciar as requisições HTTP, aplicar lógicas de negócio, interagir com o banco de dados através do Sequelize ORM e garantir a segurança da aplicação com autenticação baseada em JWT.

Arquivo Principal: `index.js`

O arquivo `index.js` serve como o ponto de entrada principal para a aplicação backend. Ele orquestra a inicialização do servidor, a configuração dos middlewares essenciais, o carregamento das rotas e a conexão com o banco de dados.

Inicialização e Configuração do Express

O processo começa com a importação das dependências necessárias, incluindo o `express` para criar o servidor web e o `path` para manipulação de caminhos de arquivos. Uma instância do Express é criada (`const app = express();`), que será utilizada para configurar toda a aplicação.

```
 const express = require('express');
const path = require('path');
const app = express();
const fs = require('fs'); // Usado para verificações de arquivos
```

Configuração para Ambientes Serverless (Vercel)

A aplicação detecta se está rodando em um ambiente serverless como o Vercel através da variável de ambiente `VERCEL` . Isso permite adaptar o comportamento, como a forma de inicialização do servidor e a sincronização do banco dedados.

```
const isVercel = process.env.VERCEL === '1';
const isProduction = process.env.NODE_ENV === 'production';
```

Middlewares Essenciais

Vários middlewares são configurados na sequência para preparar a aplicação para receber requisições:

1. **`express.json()`** : Este middleware é fundamental para que a aplicação consiga interpretar o corpo das requisições que chegam no formato JSON. Ele popula o objeto `req.body` com os dados parseados.

   `javascript app.use(express.json());`

2. **Servir Arquivos Estáticos (`express.static`)**: O backend também é responsável por servir os arquivos estáticos do frontend (HTML, CSS, JavaScript, imagens) localizados na pasta `public` . A configuração `express.static` é utilizada para este fim. Opções como `maxAge` , `etag` e `lastModified` são desativadas para evitar problemas de cache durante o desenvolvimento e garantir que as versões mais recentes dos arquivos sejam sempre entregues. Um middleware customizado é adicionado antes para logar requisições de arquivos estáticos, auxiliando na depuração.

   ``javascript // Middleware de log para arquivos estáticos app.use((req, res, next) => { if (req.url.match(/\.(css|jpg|jpeg|png|gif|js|ico)$/)) { console.log([DEBUG] Requisição de arquivo estático: ${req.url}`); } next(); });

   // Servir arquivos estáticos app.use(express.static(path.join(__dirname, 'public'), { maxAge: 0, etag: false, lastModified: false, index: false // Evita servir index.html automaticamente })); ```

3. **Controle de Cache para Estáticos**: Um middleware adicional é configurado especificamente para arquivos estáticos, definindo cabeçalhos HTTP (`Cache-Control` , `Pragma` , `Expires`) para instruir o navegador a não armazenar esses arquivos em cache. Isso é particularmente útil em ambientes de desenvolvimento para garantir que as alterações no frontend sejam refletidas imediatamente.

   `javascript app.use((req, res, next) => { if (req.url.match(/\.(css|jpg|jpeg|png|gif|js|ico)$/)) { res.setHeader('Cache-Control', 'no-store, no-cache, must-revalidate, proxy-revalidate'); res.setHeader('Pragma', 'no-cache'); res.setHeader('Expires', '0'); } next(); });`

Definição de Rotas Estáticas e Raiz

Rotas explícitas são definidas para servir os arquivos HTML principais da aplicação (`login.html` , `register.html` , `home.html` , `new_task.html`). A rota raiz (`/`) redireciona para a página de login.

```
app.get('/', (_, res) => {
  res.sendFile(path.join(__dirname, 'public', 'login.html'));
});

app.get('/login.html', (_, res) => { /* ... */ });
app.get('/register.html', (_, res) => { /* ... */ });
app.get('/home.html', (_, res) => { /* ... */ });
app.get('/new_task.html', (_, res) => { /* ... */ });
```

Rota de Diagnóstico

Uma rota `/debug/css-check` foi adicionada para auxiliar na depuração de problemas relacionados ao carregamento de arquivos CSS, verificando a existência e o tamanho dos arquivos CSS esperados.

```
app.get('/debug/css-check', (req, res) => { /* ... lógica de verificação ... */ });
```

Inicialização do Banco de Dados e Rotas da API

A configuração e inicialização do banco de dados (Sequelize) são importadas de `./config/index.js` . As rotas da API, que dependem do banco de dados (`/auth` e `/tarefas`), são carregadas e associadas aos seus respectivos prefixos. A sincronização do banco de dados (`database.sync`) é tentada apenas em ambiente local (não Vercel), garantindo que as tabelas sejam criadas ou atualizadas conforme os modelos definidos.

```
let database;
try {
  database = require('./config/index.js');

  const tarefasRoutes = require('./routes/tarefas.js');
  const authRoutes = require('./routes/auth.js');

  app.use('/auth', authRoutes);
  app.use('/tarefas', tarefasRoutes);

  if (!isVercel) {
    database.sync({ force: false }).then(() => { /* ... */ }).catch(err => { /* ... */ });
  }
```

```
} catch (err) {
  console.error('Erro ao inicializar banco de dados:', err);
}
```

Tratamento de Erros

Dois middlewares finais são configurados para tratamento de erros:

1. **Rota Não Encontrada (404)**: Captura qualquer requisição que não corresponda a uma rota definida (API ou página HTML) e retorna uma resposta JSON com status 404. Requisições para arquivos estáticos não encontrados passam para o próximo handler.

   `javascript app.use((req, res, next) => { // ... lógica para diferenciar arquivos estáticos de rotas API/HTML ... res.status(404).json({ mensagem: 'Endpoint não encontrado' }); });`

2. **Erro Interno do Servidor (500)**: Captura quaisquer erros que ocorram durante o processamento das requisições. Loga o erro no console e retorna uma resposta JSON com status 500. Em produção, a mensagem de erro detalhada é omitida por segurança.

   `javascript app.use((err, req, res, next) => { console.error('Erro na aplicação:', err); // ... lógica para diferenciar arquivos estáticos de rotas API/HTML ... res.status(500).json({ mensagem: 'Erro interno do servidor', erro: process.env.NODE_ENV === 'production' ? null : err.message }); });`

Inicialização do Servidor

Finalmente, a lógica de inicialização do servidor difere entre o ambiente local e o Vercel. Localmente, `app.listen` é chamado para iniciar o servidor na porta definida pela variável de ambiente `PORT` ou, por padrão, na porta 8080, escutando em `0.0.0.0` para permitir acesso externo. No Vercel, a instância `app` é exportada para ser gerenciada pela plataforma serverless.

```
if (isVercel) {
  console.log('Executando em ambiente Vercel');
  module.exports = app;
} else {
  const PORT = process.env.PORT || 8080;
  app.listen(PORT, '0.0.0.0', () => {
    console.log(`Servidor rodando na porta ${PORT}`);
```

```
  });
}
```

Este arquivo `index.js` estabelece a fundação da aplicação backend, configurando o servidor, middlewares, rotas e tratamento de erros de forma organizada e adaptável a diferentes ambientes de execução.

Rotas de Autenticação: `routes/auth.js`

Este arquivo é responsável por definir as rotas específicas para as funcionalidades de autenticação de usuários na aplicação Taskea. Ele utiliza o `express.Router` para criar um módulo de rotas dedicado, mantendo o código organizado e modular.

Configuração do Router

Primeiramente, o `express` é importado e uma instância do `Router` é criada. Em seguida, o `authController`, que contém a lógica de negócio para registro e login, é importado do diretório `../controllers/`.

```
const express = require("express");
const router = express.Router();
const authController = require("../controllers/authController.js");
```

Definição das Rotas

Duas rotas são definidas neste módulo, ambas utilizando o método HTTP POST, adequado para enviar dados sensíveis como credenciais de usuário:

1. **`/registrar` (POST)**: Esta rota é designada para o registro de novos usuários. Quando uma requisição POST é feita para `/auth/registrar` (lembrando que o prefixo `/auth` é definido em `index.js`), a função `registrarUsuario` do `authController` é invocada para processar a requisição. Espera-se que o corpo da requisição contenha os dados necessários para o cadastro (como nome, email e senha).

   `javascript router.post("/registrar", authController.registrarUsuario);`

2. **`/login` (POST)**: Esta rota é utilizada para autenticar usuários existentes. Uma requisição POST para `/auth/login` aciona a função `loginUsuario` no `authController`. O corpo da requisição deve incluir as credenciais do usuário (email e senha) para validação.

`javascript router.post("/login", authController.loginUsuario);`

Exportação do Router

Finalmente, o objeto `router` configurado é exportado para que possa ser importado e utilizado no arquivo principal da aplicação (`index.js`), onde será montado sob o prefixo `/auth` .

```
module.exports = router;
```

Este arquivo atua como um direcionador, conectando os endpoints de autenticação expostos pela API à lógica de controle correspondente, promovendo uma separação clara de responsabilidades dentro da arquitetura do backend.

Controller de Autenticação: `controllers/authController.js`

Este controller encapsula a lógica de negócio relacionada à autenticação de usuários, incluindo o registro de novas contas e o processo de login. Ele interage diretamente com o modelo `Usuario` para persistir e verificar dados, além de utilizar a biblioteca `jsonwebtoken` para gerar tokens de acesso.

Dependências

O controller importa o modelo `Usuario` para interagir com a tabela de usuários no banco de dados e a biblioteca `jsonwebtoken` para a criação de tokens JWT.

```
const Usuario = require('../models/Usuario.js');
const jwt = require('jsonwebtoken');
```

Função `registrarUsuario`

Esta função assíncrona é responsável por lidar com as requisições de registro de novos usuários (`POST /auth/registrar`).

Fluxo de Execução:

1. **Extração de Dados**: Os dados do novo usuário (`nome` , `email` , `senha`) são extraídos do corpo da requisição (`req.body`).

2. **Validação de Entrada**: Verifica se todos os campos obrigatórios (`nome` , `email` , `senha`) foram fornecidos. Se algum estiver faltando, retorna um erro 400 (Bad Request) com uma mensagem informativa.
3. **Verificação de Email Existente**: Consulta o banco de dados usando `Usuario.findOne` para verificar se já existe um usuário cadastrado com o email fornecido. Se o email já estiver em uso, retorna um erro 400 indicando que o email já está cadastrado.
4. **Criação do Usuário**: Se o email não existir e os dados forem válidos, um novo usuário é criado no banco de dados utilizando `Usuario.create(req.body)` . O Sequelize automaticamente lida com o hashing da senha, conforme definido no modelo `Usuario` (utilizando `bcrypt` internamente através de um hook `beforeSave`).
5. **Resposta de Sucesso**: Em caso de sucesso, a função retorna o status 201 (Created) junto com um objeto JSON contendo o `id` , `nome` e `email` do usuário recém-criado (a senha é omitida por segurança).

```
exports.registrarUsuario = async (req, res) => {
  const {nome, email, senha} = req.body;

  if (!nome || !email || !senha) {
    return res.status(400).json({ mensagem: 'Preencha nome, email e senha.' });
  }
  const usuarioExistente = await Usuario.findOne({ where: { email: email } });

  if (usuarioExistente) {
    return res.status(400).json({mensagem: 'Email já cadastrado.'})
  }

  // A senha será hasheada automaticamente pelo hook do modelo Usuario
  const usuario = await Usuario.create(req.body);

  const {id} = usuario;

  return res.status(201).json({id, nome, email});
};
```

Função `loginUsuario`

Esta função assíncrona gerencia o processo de login de usuários existentes (`POST /auth/login`).

Fluxo de Execução:

1. **Extração de Credenciais**: O `email` e a `senha` são extraídos do corpo da requisição (`req.body`).
2. **Validação de Entrada**: Verifica se ambos os campos (`email`, `senha`) foram fornecidos. Se algum estiver faltando, retorna um erro 400.
3. **Busca do Usuário**: Procura por um usuário no banco de dados que corresponda ao `email` fornecido usando `Usuario.findOne`.
4. **Verificação de Usuário**: Se nenhum usuário for encontrado com o email fornecido, retorna um erro 401 (Unauthorized) com uma mensagem genérica ("Email ou senha inválidos") para evitar a enumeração de usuários.
5. **Verificação de Senha**: Se o usuário for encontrado, o método `checkPassword(senha)` (definido no modelo `Usuario`) é chamado. Este método compara a senha fornecida com o hash armazenado no banco de dados usando `bcrypt.compare`. Se a senha não corresponder, retorna um erro 401.
6. **Geração do Token JWT**: Se a senha estiver correta, um token JWT é gerado usando `jwt.sign`. O payload do token inclui o `id`, `email` e `nome` do usuário. Um segredo (`'Seguro'`) é usado para assinar o token (em um ambiente de produção, este segredo deve ser mais complexo e armazenado de forma segura, por exemplo, em variáveis de ambiente). O token tem um tempo de expiração definido para 1 hora (`expiresIn: '1h'`).
7. **Resposta de Sucesso**: Retorna o status 200 (OK) com uma mensagem de sucesso, o token JWT gerado e o nome do usuário.

```
exports.loginUsuario = async (req, res) => {
  const {email, senha} = req.body;

  if (!email || !senha) {
    return res.status(400).json({mensagem: 'Preencha os campos de email e
senha.'})
  }

  const usuario = await Usuario.findOne({ where: { email: email } });

  if (!usuario) {
    return res.status(401).json({mensagem: 'Email ou senha inválidos'})
  }

  // O método checkPassword usa bcrypt.compare internamente
  if (!(await usuario.checkPassword(senha))) {
    return res.status(401).json({mensagem: 'Email ou senha inválidos.'})
  }

  const {id, nome} = usuario;
```

```
    // O segredo 'Seguro' deve ser substituído por uma variável de ambiente em
produção
    const token = jwt.sign(
        {id: id, email: email, nome: nome}, 'Seguro',
        {expiresIn: '1h'}
    );

    return res.status(200).json({
        mensagem: 'Login realizado com sucesso',
        token,
        nome: nome
    });
};
```

Este controller implementa a lógica central de autenticação, garantindo a validação de dados, a segurança das senhas e a emissão de tokens para sessões autenticadas.

Rotas de Tarefas: `routes/tarefas.js`

Este arquivo define as rotas para o gerenciamento de tarefas (operações CRUD - Create, Read, Update, Delete) dentro da aplicação Taskea. Assim como as rotas de autenticação, ele utiliza o `express.Router` para modularidade e aplica um middleware de autenticação para garantir que apenas usuários logados possam acessar e manipular suas tarefas.

Configuração do Router e Middleware

O arquivo começa importando o `express` e criando uma instância do `Router`. Crucialmente, ele importa o middleware de autenticação (`autenticacao`) de `../middlewares/authMiddleware.js` e o controller de tarefas (`tarefaController`) de `../controllers/tarefaController.js` .

```
 const express  = require('express');
const router = express.Router();
const autenticacao = require('../middlewares/authMiddleware');
const tarefaController = require('../controllers/tarefaController')
```

O middleware `autenticacao` é aplicado a todas as rotas definidas neste arquivo. Isso significa que, antes de qualquer função do `tarefaController` ser executada, o middleware verificará a validade do token JWT presente no cabeçalho da requisição. Se o token for válido, o middleware adicionará as informações do usuário (como `id` e `nome`) ao objeto `req` (geralmente `req.usuario`), tornando-as disponíveis para os

controllers. Se o token for inválido ou ausente, o middleware retornará um erro de não autorizado (401), impedindo o acesso à rota.

Definição das Rotas de Tarefas

As seguintes rotas são definidas, todas prefixadas com `/tarefas` (conforme configurado em `index.js`) e protegidas pelo middleware `autenticacao`:

1. **`/hello` (GET)**: Uma rota simples para teste e boas-vindas. Ela utiliza o middleware de autenticação para obter o nome do usuário logado (`req.usuario.nome`) e retorna uma mensagem personalizada. Isso demonstra o funcionamento básico da autenticação e o acesso aos dados do usuário injetados pelo middleware.

   `javascript router.get('/hello', autenticacao, (req, res) => { res.json({mensagem: `Bem vindo ${req.usuario.nome}! Aqui estão suas tarefas disponíveis.`}) });`

2. **`/` (POST)**: Rota para criar uma nova tarefa. Uma requisição POST para `/tarefas` aciona a função `criarTarefa` no `tarefaController`. O corpo da requisição deve conter os dados da nova tarefa (por exemplo, título, descrição). O middleware garante que apenas um usuário autenticado possa criar tarefas, e o `id` do usuário (obtido de `req.usuario.id`) será associado à tarefa criada.

   `javascript router.post('/', autenticacao, tarefaController.criarTarefa)`

3. **`/` (GET)**: Rota para listar todas as tarefas pertencentes ao usuário autenticado. Uma requisição GET para `/tarefas` chama a função `listarTarefas` do `tarefaController`. O controller utilizará o `id` do usuário (de `req.usuario.id`) para filtrar e retornar apenas as tarefas associadas a ele.

   `javascript router.get('/', autenticacao, tarefaController.listarTarefas)`

4. **`/:id` (PUT)**: Rota para atualizar uma tarefa existente. Uma requisição PUT para `/tarefas/:id` (onde `:id` é o ID da tarefa a ser atualizada) invoca a função `editarTarefa` no `tarefaController`. O corpo da requisição deve conter os dados atualizados da tarefa. O controller verificará se a tarefa pertence ao usuário autenticado antes de permitir a edição.

   `javascript router.put('/:id', autenticacao, tarefaController.editarTarefa)`

5. **`/:id` (DELETE)**: Rota para excluir uma tarefa existente. Uma requisição DELETE para `/tarefas/:id` (onde `:id` é o ID da tarefa a ser excluída) chama a função `deletarTarefa` do `tarefaController`. Assim como na edição, o controller garantirá

que o usuário autenticado seja o proprietário da tarefa antes de prosseguir com a exclusão.

`javascript router.delete('/:id', autenticacao, tarefaController.deletarTarefa)`

Exportação do Router

O router configurado com todas as rotas de tarefas é exportado para ser utilizado no `index.js`.

```
module.exports = router;
```

Este arquivo organiza de forma eficaz as rotas relacionadas às tarefas, aplicando consistentemente a autenticação e delegando a lógica de negócio específica para o `tarefaController`, seguindo as melhores práticas de desenvolvimento de APIs RESTful.

Controller de Tarefas: `controllers/tarefaController.js`

Este controller é o núcleo da funcionalidade de gerenciamento de tarefas da aplicação Taskea. Ele contém a lógica para criar, listar, editar e excluir tarefas, interagindo diretamente com o modelo `Tarefa` do Sequelize para realizar operações no banco de dados. Todas as operações neste controller são acessadas através das rotas definidas em `routes/tarefas.js` e, portanto, protegidas pelo middleware de autenticação.

Dependências

O controller depende exclusivamente do modelo `Tarefa` para acessar e manipular os dados das tarefas no banco de dados.

```
const Tarefa = require('../models/Tarefa.js');
```

Função `criarTarefa`

Responsável por processar requisições `POST /tarefas` para a criação de novas tarefas.

Fluxo de Execução:

1. **Extração de Dados**: O `titulo` e a `descricao` da nova tarefa são extraídos do corpo da requisição (`req.body`). O campo `usuario_id` também é extraído, embora o código atual não o utilize explicitamente para associar a tarefa ao usuário logado (o middleware `autenticacao` injeta `req.usuario`, que deveria ser usado aqui para

definir o `usuario_id` da tarefa, garantindo que a tarefa pertença ao usuário correto. A implementação atual permite que o frontend envie qualquer `usuario_id`, o que é uma falha de segurança e lógica).

2. **Validação de Entrada**: Verifica se os campos `titulo` e `descricao` foram fornecidos. Se algum estiver ausente, retorna um erro 400 (Bad Request) com uma mensagem indicando os campos obrigatórios.
3. **Criação da Tarefa**: Utiliza `Tarefa.create(req.body)` para inserir uma nova tarefa no banco de dados. O Sequelize mapeia os campos do `req.body` para as colunas da tabela `Tarefas`. Observação: Idealmente, deveria ser `Tarefa.create({ titulo, descricao, usuario_id: req.usuario.id })` para garantir a associação correta e segura ao usuário autenticado.
4. **Resposta de Sucesso**: Retorna o status 201 (Created) com uma mensagem de sucesso e o objeto da tarefa recém-criada (incluindo o ID gerado pelo banco de dados e outros campos definidos no modelo).

```
exports.criarTarefa = async (req, res) => {
  const {titulo, descricao, usuario_id} = req.body; // usuario_id recebido do body é problemático

  if (!titulo || !descricao) {
    return res.status(400).json({ mensagem: 'Título e descrição são obrigatórios.' });
  }

  // Idealmente: const tarefa = await Tarefa.create({ titulo, descricao, usuario_id: req.usuario.id });
  const tarefa = await Tarefa.create(req.body);

  res.status(201).json({ mensagem: 'Tarefa criada com sucesso.', tarefa});
};
```

Função `listarTarefas`

Responsável por processar requisições `GET /tarefas` para buscar e retornar as tarefas.

Fluxo de Execução:

1. **Busca de Tarefas**: Utiliza `Tarefa.findAll()` para buscar todas as tarefas no banco de dados. Observação: A implementação atual busca todas as tarefas do banco, independentemente do usuário logado. Deveria filtrar as tarefas pelo `usuario_id` do usuário autenticado: `Tarefa.findAll({ where: { usuario_id: req.usuario.id }, attributes: { exclude: ['UsuarioId'] } })`.

2. **Exclusão de Atributo**: A opção `attributes: {exclude: ['UsuarioId']}` é usada para omitir o campo `UsuarioId` (chave estrangeira) da resposta, o que é uma boa prática para não expor detalhes internos da associação.
3. **Resposta**: Retorna um array JSON contendo todas as tarefas encontradas (ou, idealmente, apenas as do usuário logado).

```
exports.listarTarefas = async (req, res) => {
  // Idealmente: const tarefasDoUsuario = await Tarefa.findAll({ where: { usuario_id: req.usuario.id }, attributes: {exclude: ['UsuarioId']} });
  const tarefasDoUsuario = await Tarefa.findAll({
    attributes: {exclude: ['UsuarioId']},
  });
  res.json(tarefasDoUsuario);
};
```

Função `editarTarefa`

Responsável por processar requisições `PUT /tarefas/:id` para atualizar uma tarefa existente.

Fluxo de Execução:

1. **Extração de Dados**: O `id` da tarefa é extraído dos parâmetros da rota (`req.params.id`). Os dados a serem atualizados (`titulo`, `descricao`, `status`) são extraídos do corpo da requisição (`req.body`).
2. **Busca da Tarefa**: Utiliza `Tarefa.findByPk(id)` para encontrar a tarefa específica pelo seu ID. A opção `attributes: {exclude: ['UsuarioId']}` é usada aqui também, embora não seja estritamente necessária para a lógica de atualização.
3. **Verificação de Existência e Autorização**: Verifica se a tarefa foi encontrada (`!tarefa`). Se não encontrada, retorna um erro 404 (Not Found). Observação: A implementação atual não verifica se a tarefa encontrada pertence ao usuário autenticado (*tarefa.usuario_id === req.usuario.id*). Isso permite que um usuário edite tarefas de outros usuários, o que é uma falha de segurança crítica. A verificação de propriedade deveria ser adicionada antes da atualização.
4. **Atualização da Tarefa**: Se a tarefa for encontrada (e, idealmente, pertencer ao usuário), o método `tarefa.update({titulo, descricao, status})` é chamado para aplicar as alterações no banco de dados. O Sequelize atualiza apenas os campos fornecidos no objeto.
5. **Resposta de Sucesso**: Retorna uma mensagem de sucesso e o objeto da tarefa atualizada.

```
exports.editarTarefa = async (req, res) => {
  const id = req.params.id;
  const{titulo, descricao, status} = req.body;

  const tarefa = await Tarefa.findByPk(id, {
    attributes: {exclude: ['UsuarioId']}
  });

  if (!tarefa) {
    return res.status(404).json({mensagem: 'Tarefa não encontrada.'}); // Mensagem ajustada
  }

  // !!! FALTA VERIFICAÇÃO DE PROPRIEDADE: if (tarefa.usuario_id !== req.usuario.id) { return res.status(403).json({mensagem: 'Não autorizado'}); }

  await tarefa.update({titulo, descricao, status});

  res.json({mensagem: 'Tarefa atualizada com sucesso.', tarefa});
};
```

Função `deletarTarefa`

Responsável por processar requisições `DELETE /tarefas/:id` para excluir uma tarefa.

Fluxo de Execução:

1. **Extração do ID**: O `id` da tarefa a ser excluída é obtido de `req.params.id`.
2. **Busca da Tarefa**: Utiliza `Tarefa.findByPk(id)` para localizar a tarefa.
3. **Verificação de Existência e Autorização**: Verifica se a tarefa existe (`!tarefa`). Se não, retorna um erro 404. Observação: Assim como na edição, falta a verificação crucial para garantir que a tarefa pertence ao usuário autenticado (*tarefa.usuario_id === req.usuario.id*) antes de permitir a exclusão.
4. **Exclusão da Tarefa**: Se a tarefa for encontrada (e, idealmente, pertencer ao usuário), o método `tarefa.destroy()` é chamado para removê-la do banco de dados.
5. **Resposta de Sucesso**: Retorna o status 204 (No Content), que é o padrão para exclusões bem-sucedidas, indicando que a operação foi realizada mas não há conteúdo para retornar no corpo da resposta. A mensagem JSON incluída na implementação atual (res.status(204).json(...)) tecnicamente viola o padrão HTTP para 204, que não deve ter corpo.

```
exports.deletarTarefa = async (req, res) => {
  const id = req.params.id;
```

```
    const tarefa = await Tarefa.findByPk(id, {
        // attributes: {exclude: ['UsuarioId']} // Não necessário para delete
    });

    if (!tarefa) {
        return res.status(404).json({mensagem: 'Tarefa não encontrada.'});
    }

    // !!! FALTA VERIFICAÇÃO DE PROPRIEDADE: if (tarefa.usuario_id !== req.usuario.id)
{ return res.status(403).json({mensagem: 'Não autorizado'}); }

    await tarefa.destroy();

    // Idealmente: return res.status(204).send();
    res.status(204).json({mensagem: 'Tarefa excluída com êxito.', tarefa}); // Retornar
JSON com 204 não é padrão
};
```

Este controller implementa as operações básicas de CRUD para tarefas, mas requer atenção especial às questões de autorização (verificação de propriedade da tarefa) para garantir a segurança e o isolamento dos dados entre usuários.

Middleware de Autenticação: `middlewares/authMiddleware.js`

Este middleware é uma peça central na segurança da API Taskea, responsável por proteger rotas que exigem que o usuário esteja autenticado. Ele intercepta as requisições destinadas a rotas protegidas, verifica a validade de um token JWT (JSON Web Token) fornecido no cabeçalho `Authorization` e, se o token for válido, permite que a requisição prossiga para o controller, anexando as informações do usuário autenticado ao objeto `req`.

Dependências

O middleware utiliza a biblioteca `jsonwebtoken` para verificar e decodificar os tokens JWT.

```
const jwt = require("jsonwebtoken");
```

Lógica do Middleware

O middleware é exportado como uma função que segue a assinatura padrão de middlewares do Express (`(req, res, next)`).

Fluxo de Execução:

1. **Extração do Token**: O middleware primeiro tenta extrair o token JWT do cabeçalho `Authorization` da requisição. Ele espera que o cabeçalho siga o formato `Bearer <token>`. O código `req.headers['authorization']` acessa o valor do cabeçalho, e `authHeader && authHeader.split(' ')[1]` divide a string no espaço e pega a segunda parte (o token em si), tratando o caso em que o cabeçalho pode estar ausente (`authHeader && ...`).

   `javascript const authHeader = req.headers["authorization"]; const token = authHeader && authHeader.split(" ")[1];`

2. **Verificação de Token Ausente**: Se nenhum token for encontrado (`!token`), o middleware imediatamente interrompe o fluxo da requisição e envia uma resposta com status 401 (Unauthorized) e uma mensagem indicando que o token não foi fornecido.

   `javascript if(!token) return res.status(401).json({mensagem: "Token não fornecido."});`

3. **Verificação do Token JWT**: Se um token for encontrado, a função `jwt.verify()` é utilizada para validá-lo. Esta função recebe três argumentos:

   - O token a ser verificado.
   - O segredo usado para assinar o token (`'Seguro'`). Observação: Usar um segredo fixo e simples como *`'Seguro'`* diretamente no código é uma prática insegura para produção. Este segredo deve ser complexo e armazenado de forma segura, preferencialmente em variáveis de ambiente.
   - Uma função de callback `(err, usuario)` que será executada após a tentativa de verificação.

4. **Tratamento de Erro na Verificação**: Se a verificação falhar (por exemplo, token inválido, expirado ou com assinatura incorreta), o parâmetro `err` no callback conterá um objeto de erro. Nesse caso, o middleware retorna uma resposta com status 403 (Forbidden) e uma mensagem indicando que o token é inválido ou expirou.

   `javascript jwt.verify(token, "Seguro", (err, usuario) => { if (err) { return res.status(403).json({mensagem: "Token inválido ou expirado."}); } // ... (continua se não houver erro) });`

5. **Sucesso na Verificação e Injeção do Usuário**: Se `jwt.verify()` for bem-sucedido, o parâmetro `err` será nulo, e o segundo parâmetro (`usuario` neste caso) conterá o

payload decodificado do token (que inclui `id`, `email` e `nome`, conforme definido em `authController.loginUsuario`). O middleware então anexa este payload decodificado ao objeto `req` sob a propriedade `usuario` (`req.usuario = usuario;`). Isso torna as informações do usuário autenticado facilmente acessíveis nos próximos middlewares ou nos controllers das rotas.

6. **Chamada `next()`**: Finalmente, a função `next()` é chamada sem argumentos. Isso sinaliza ao Express que o middleware concluiu seu processamento e que a requisição pode continuar para o próximo middleware na cadeia ou para o handler da rota final (o controller).

   `javascript // ... (dentro do callback de jwt.verify, após a verificação de erro) req.usuario = usuario; next();`

Exportação

O middleware configurado é exportado para ser utilizado na definição das rotas protegidas (como visto em `routes/tarefas.js`).

```
module.exports = (req, res, next) => { /* ... lógica do middleware ... */ };
```

Este middleware implementa um mecanismo de autenticação baseado em token JWT padrão, essencial para proteger os recursos da API Taskea, garantindo que apenas usuários com tokens válidos possam acessar as funcionalidades de gerenciamento de tarefas.

Modelos de Dados (Sequelize)

Os modelos de dados definem a estrutura das tabelas no banco de dados e fornecem uma interface orientada a objetos para interagir com esses dados. O Taskea utiliza o Sequelize ORM, e os modelos são definidos como classes que estendem `Sequelize.Model`.

Modelo `Usuario`: `models/Usuario.js`

Este modelo representa a entidade "Usuário" na aplicação e mapeia para a tabela `usuarios` no banco de dados. Ele define os atributos do usuário, validações, relacionamentos e lógica específica, como o hashing de senhas.

Dependências:

O modelo importa `DataTypes`, `Model` e `Sequelize` da biblioteca `sequelize` e `bcryptjs` para hashing de senhas.

```
const { DataTypes, Model, Sequelize } = require("sequelize");
const bcrypt = require("bcryptjs");
```

Definição da Classe e Inicialização (`init`)

A classe `Usuario` estende `Sequelize.Model`. O método estático `init` é responsável por definir os atributos (colunas) da tabela e configurar o modelo.

```
class Usuario extends Model {
  static init(sequelize) {
    super.init({
      // Definição dos atributos...
    }, {
      sequelize, // Instância do Sequelize
      modelName: "Usuario", // Nome do modelo
      tableName: "usuarios" // Nome da tabela no BD
    });
    // Hooks e associações...
    return this;
  }
  // Métodos de instância e estáticos...
}
```

Atributos Definidos:

- **`id`**: Chave primária da tabela, do tipo `INTEGER`, autoincrementável. `javascript id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },`
- **`nome`**: Nome do usuário, do tipo `STRING`, não pode ser nulo (`allowNull: false`). `javascript nome: { type: DataTypes.STRING, allowNull: false },`
- **`email`**: Email do usuário, do tipo `STRING`, não pode ser nulo e deve ser único na tabela (`unique: true`). O Sequelize adicionará automaticamente um índice único para esta coluna. `javascript email: { type: DataTypes.STRING, allowNull: false, unique: true },`
- **`senha`**: Este é um campo **virtual** (`type: Sequelize.VIRTUAL`). Campos virtuais não são persistidos no banco de dados. Ele serve como um campo temporário para receber a senha em texto plano vinda da requisição (por exemplo, durante o registro ou login). O valor deste campo será usado pelo hook `beforeSave` para gerar o hash da senha. `javascript senha: { type: Sequelize.VIRTUAL },`
- **`senha_hash`**: Armazena o hash da senha do usuário, do tipo `STRING`. Este é o campo que é efetivamente salvo no banco de dados, garantindo que a senha

original nunca seja armazenada. `javascript senha_hash: { type: DataTypes.STRING, },`

Hooks (`beforeSave`)

Os hooks do Sequelize permitem executar lógica personalizada em diferentes pontos do ciclo de vida de um modelo. O hook `beforeSave` é adicionado para garantir que a senha seja hasheada antes de qualquer operação de criação (`create`) ou atualização (`update`) de um usuário.

1. O hook verifica se o campo virtual `senha` foi fornecido (`if (usuario.senha)`). Isso garante que o hash só seja recalculado se uma nova senha estiver sendo definida.
2. Se uma senha foi fornecida, `bcrypt.hash(usuario.senha, 8)` é chamado para gerar o hash. O segundo argumento (`8`) é o número de rounds de salt (custo computacional); um valor maior aumenta a segurança mas também o tempo de processamento.
3. O hash resultante é atribuído ao campo `senha_hash` , que será persistido no banco de dados.

```
this.addHook("beforeSave", async usuario => {
  if (usuario.senha) {
    usuario.senha_hash = await bcrypt.hash(usuario.senha, 8);
  }
});
```

Método de Instância (`checkPassword`)

Este método é adicionado à instância do modelo `Usuario` para facilitar a verificação de senhas durante o processo de login. Ele recebe uma senha em texto plano como argumento e a compara com o `senha_hash` armazenado para aquele usuário, utilizando `bcrypt.compare` .

`bcrypt.compare` é uma função assíncrona (embora usada de forma síncrona aqui, o que pode ser um erro dependendo da versão do bcryptjs, idealmente deveria ser `async checkPassword(senha) { return await bcrypt.compare(senha, this.senha_hash); }`) que compara a senha fornecida com o hash armazenado, retornando `true` se corresponderem e `false` caso contrário.

```
checkPassword(senha) {
  // Idealmente: return await bcrypt.compare(senha, this.senha_hash);
  return bcrypt.compare(senha, this.senha_hash);
}
```

Associações (`associate`)

O método estático `associate` é um local padrão no Sequelize para definir relacionamentos entre modelos. Aqui, ele define que um `Usuario` pode ter muitas `Tarefas` (`this.hasMany(models.Tarefa)`). Isso cria uma chave estrangeira `UsuarioId` (ou similar, dependendo da configuração) na tabela `Tarefas` , ligando cada tarefa ao seu usuário correspondente.

```
static associate(models) {
  this.hasMany(models.Tarefa);
}
```

Exportação

A classe `Usuario` configurada é exportada para ser utilizada em outros lugares da aplicação, como nos controllers e na inicialização do Sequelize.

```
module.exports = Usuario;
```

Este modelo `Usuario` encapsula de forma eficaz a estrutura de dados do usuário, a lógica de segurança de senha e o relacionamento com as tarefas, facilitando a interação com os dados do usuário de forma segura e organizada.

Modelo `Tarefa` : `models/Tarefa.js`

Este modelo representa a entidade "Tarefa" na aplicação, mapeando para a tabela `tarefas` no banco de dados. Ele define os atributos de uma tarefa, como título, descrição e status, e estabelece o relacionamento com o usuário proprietário.

Dependências:

Similar ao modelo `Usuario` , ele importa `DataTypes` , `Model` e `Sequelize` da biblioteca `sequelize` .

```
const { DataTypes, Model, Sequelize } = require("sequelize");
```

Definição da Classe e Inicialização (`init`)

A classe `Tarefa` estende `Sequelize.Model` , e o método estático `init` define seus atributos e configurações.

```
class Tarefa extends Model {
  static init(sequelize) {
    super.init({
      // Definição dos atributos...
    }, {
      sequelize,
      modelName: "Tarefa", // Nome do modelo
      tableName: "tarefas" // Nome da tabela no BD
    });
    // Associações...
    return this;
  }
  // Métodos...
}
```

Atributos Definidos:

- **id** : Chave primária da tabela, do tipo `INTEGER` , autoincrementável. `javascript id: { type: Sequelize.INTEGER, primaryKey: true, autoIncrement: true },`
- **titulo** : Título da tarefa, do tipo `STRING` , não pode ser nulo (`allowNull: false`). `javascript titulo: { type: DataTypes.STRING, allowNull: false },`
- **descricao** : Descrição detalhada da tarefa, do tipo `STRING` , não pode ser nula (`allowNull: false`). `javascript descricao: { type: DataTypes.STRING, allowNull: false },`
- **status** : Status atual da tarefa. É definido como um tipo `ENUM` , que restringe os valores possíveis a um conjunto pré-definido: `'pendente'` , `'em andamento'` , `'concluida'` . Isso garante a consistência dos dados de status. Possui um valor padrão (`defaultValue: 'pendente'`) e não pode ser nulo (`allowNull: false`). `javascript status: { type: DataTypes.ENUM('pendente', 'em andamento', 'concluida'), defaultValue: 'pendente', allowNull: false }`

Associações (associate)

O método estático `associate` define o relacionamento entre `Tarefa` e `Usuario` . A linha `this.belongsTo(models.Usuario, { ... })` estabelece que cada `Tarefa` pertence a um `Usuario` .

- **foreignKey: 'usuario_id'** : Especifica que a chave estrangeira na tabela `tarefas` que referencia a tabela `usuarios` se chamará `usuario_id` . O Sequelize criará esta coluna automaticamente.
- **onDelete: 'CASCADE'** : Esta é uma configuração crucial do relacionamento. Ela instrui o banco de dados a **excluir automaticamente todas as tarefas associadas**

a um usuário se esse usuário for excluído da tabela `usuarios`. Isso mantém a integridade referencial, evitando tarefas órfãs.

```
  static associate(models) {
    this.belongsTo(models.Usuario, {
      foreignKey: 'usuario_id',
      onDelete: 'CASCADE',
    });
  }
```

Exportação

A classe `Tarefa` configurada é exportada.

```
module.exports = Tarefa;
```

O modelo `Tarefa` define claramente a estrutura dos dados de tarefas e seu vínculo essencial com os usuários, incluindo a importante regra de exclusão em cascata para manutenção da consistência do banco de dados.