Testes Unitários e Nova Análise com SonarQube

Objetivo da Atividade

Implementar **testes unitários com Jest** para os arquivos userController.js e userService.js, executar os testes, enviar os resultados ao **SonarQube**, e analisar os indicadores de **qualidade e cobertura de código**.

Exemplos de Testes Criados:

Teste unitários em userController.js:
Função para listar usuário.

```
const getUserById = async (req, res) => {
    try {
        const user = await userModel.getUserById(parseInt(req.params.id));
        if (!user) return res.status(404).json({ error: 'Usuário não encontrado.' });
        res.status(200).json(user);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
};
```

```
const { getUserById } = require('./userController');
const userModel = require('../models/userModel');

// Mock do userModel
jest.mock('../models/userModel');

describe('getUserById', () => {
  let req, res;

  beforeEach(() => {
    req = {
      params: { id: '123' },
    };
```

```
    res = {
      status: jest.fn().mockReturnThis(),
      json: jest.fn(),
    };
  });

  afterEach(() => {
    jest.clearAllMocks();
  });

  it('deve retornar 200 e o usuário quando encontrado', async () => {
    const mockUser = { id: 123, name: 'John Doe' };
    userModel.getUserById.mockResolvedValue(mockUser);

    await getUserById(req, res);

    expect(userModel.getUserById).toHaveBeenCalledWith(123);
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.json).toHaveBeenCalledWith(mockUser);
  });

  it('deve retornar 404 quando o usuário não é encontrado', async () => {
    userModel.getUserById.mockResolvedValue(null);

    await getUserById(req, res);

    expect(res.status).toHaveBeenCalledWith(404);
    expect(res.json).toHaveBeenCalledWith({ error: 'Usuário não encontrado.' });
  });

  it('deve retornar 500 em caso de erro interno', async () => {
    const errorMessage = 'Erro interno no servidor';
    userModel.getUserById.mockRejectedValue(new Error(errorMessage));

    await getUserById(req, res);

    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({ error: errorMessage });
  });
});
```

Usamos `jest.mock` para simular o modelo `userModel` e controlar o que ele retorna.
Criamos `req` e `res` como objetos simulados, com os métodos que a função usa (`res.status`, `res.json`).
Cada teste verifica um cenário diferente:
Usuário encontrado → 200 OK
Usuário não encontrado → 404
Erro interno → 500

2º Teste com userController.js:
Função para deletar usuario.

```
const deleteUser = async (req, res) => {
    try {
        await userModel.deleteUser(parseInt(req.params.id));
        res.status(204).send();
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
};
```

```
const { deleteUser } = require('./userController');
const userModel = require('../models/userModel');

// Mock do userModel
jest.mock('../models/userModel');

describe('deleteUser', () => {
  let req, res;

  beforeEach(() => {
    req = {
      params: { id: '123' },
    };
    res = {
      status: jest.fn().mockReturnThis(),
      send: jest.fn(),
      json: jest.fn(),
    };
  });

  afterEach(() => {
    jest.clearAllMocks();
  });

  it('deve retornar 204 após deletar o usuário com sucesso', async () => {
    userModel.deleteUser.mockResolvedValue();

    await deleteUser(req, res);

    expect(userModel.deleteUser).toHaveBeenCalledWith(123);
    expect(res.status).toHaveBeenCalledWith(204);
    expect(res.send).toHaveBeenCalled();
  });

  it('deve retornar 500 em caso de erro interno', async () => {
    const errorMessage = 'Erro ao deletar usuário';
    userModel.deleteUser.mockRejectedValue(new Error(errorMessage));

    await deleteUser(req, res);

    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({ error: errorMessage });
  });
});
```

Explicação:

O teste simula o comportamento do `userModel.deleteUser`, tanto para sucesso quanto para erro.
Em caso de sucesso, a função deve retornar status `204` e chamar `res.send()`.
Em caso de erro, deve retornar status `500` com a mensagem de erro em JSON.

Agora Analisaremos 2 trechos dos codigos do userService.js com os testes unitarios:
Função para criar um novo usuário.

```
const createUser = (userData) => {
    const users = getAllUsers();
    const newId = users.length > 0 ? Math.max(...users.map(user => user.id)) + 1 : 1;
    const newUser = { id: newId, ...userData };
    users.push(newUser);
    saveUsers(users);
    return newUser;
};
```

```
const { createUser } = require('./userServicer');
const { getAllUsers, saveUsers } = require('../models/userModel');

// Mock do modelo
jest.mock('../models/userModel');

describe('createUser', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('deve criar um novo usuário com ID sequencial quando há usuários existentes', () => {
    const mockUsers = [
      { id: 1, name: 'Alice', email: 'alice@example.com' },
      { id: 2, name: 'Bob', email: 'bob@example.com' },
    ];
    const newUserInput = { name: 'Charlie', email: 'charlie@example.com' };
    const expectedNewUser = { id: 3, name: 'Charlie', email: 'charlie@example.com' };

    getAllUsers.mockReturnValue(mockUsers);
    saveUsers.mockReturnValue();

    const result = createUser(newUserInput);

    expect(getAllUsers).toHaveBeenCalled();
    expect(result).toEqual(expectedNewUser);
    expect(saveUsers).toHaveBeenCalledWith([...mockUsers, expectedNewUser]);
  });

  it('deve criar um novo usuário com ID 1 quando não há usuários existentes', () => {
    const mockUsers = [];
    const newUserInput = { name: 'Dave', email: 'dave@example.com' };
    const expectedNewUser = { id: 1, name: 'Dave', email: 'dave@example.com' };

    getAllUsers.mockReturnValue(mockUsers);
    saveUsers.mockReturnValue();

    const result = createUser(newUserInput);

    expect(getAllUsers).toHaveBeenCalled();
    expect(result).toEqual(expectedNewUser);
```

```
    expect(saveUsers).toHaveBeenCalledWith([expectedNewUser]);
  });
```

Explicação:
O teste simula o comportamento de getAllUsers e saveUsers usando jest.fn().
Verifica se o ID é gerado corretamente com base no maior ID existente.
Confirma que o novo usuário é salvo corretamente na lista de usuários.
Se a função createUser for assíncrona ou usar funções assíncronas internamente, a estrutura do teste pode mudar um pouco. Se quiser adaptar para o caso assíncrono, posso te ajudar também.

2º Teste userService.js:
Função para atualizar usuário.

```
const updateUser = (id, updatedData) => {
    const users = getAllUsers();
    const userIndex = users.findIndex(user => user.id === id);
    if (userIndex === -1) throw new Error('Usuário não encontrado.');
    users[userIndex] = { ...users[userIndex], ...updatedData };
    saveUsers(users);
    return users[userIndex];
};
```

```
const { updateUser } = require('./userService');
const { getAllUsers, saveUsers } = require('../models/userModel');

// Mock do modelo
jest.mock('../models/userModel');

describe('updateUser', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('deve atualizar o usuário e salvar quando o ID existe', () => {
    const mockUsers = [
      { id: 1, name: 'Alice', email: 'alice@example.com' },
      { id: 2, name: 'Bob', email: 'bob@example.com' },
    ];
    const idToUpdate = 1;
    const updatedData = { name: 'Alice Updated', email: 'alice.updated@example.com' };
    const expectedUpdatedUser = { id: 1, name: 'Alice Updated', email:
'alice.updated@example.com' };

    getAllUsers.mockReturnValue(mockUsers);
    saveUsers.mockReturnValue();

    const result = updateUser(idToUpdate, updatedData);

    expect(getAllUsers).toHaveBeenCalled();
    expect(result).toEqual(expectedUpdatedUser);
    expect(saveUsers).toHaveBeenCalledWith([
      expectedUpdatedUser,
      { id: 2, name: 'Bob', email: 'bob@example.com' },
    ]);
  });

  it('deve lançar um erro quando o usuário não é encontrado', () => {
    const mockUsers = [
```

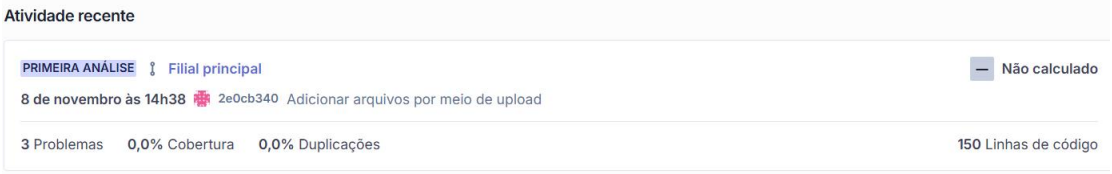
```
      { id: 1, name: 'Alice', email: 'alice@example.com' },
      { id: 2, name: 'Bob', email: 'bob@example.com' },
    ];
    const idToUpdate = 999;
    const updatedData = { name: 'Nonexistent' };

    getAllUsers.mockReturnValue(mockUsers);

    expect(() => {
      updateUser(idToUpdate, updatedData);
    }).toThrow('Usuário não encontrado.');
    expect(saveUsers).not.toHaveBeenCalled();
  });
});
```

Explicação:
O primeiro teste simula a atualização de um usuário existente e verifica se saveUsers é chamado com a lista atualizada.
O segundo teste verifica que um erro é lançado quando o ID não existe e que saveUsers não é chamado nesse caso.

Ao tentar Rodar o codigo no SonarQube para os demais resultados , ainda com bastante dificuldade para fazer os teste obtive as seguintes informações do código:



E ainda:

Mais:

Tentei de varias maneiras trazer mais resultados para os teste no SonarQube porém parece que não obtive sucesso na analise geral do código, apresentando essa mensagem:

Como se não tivesse calculando todos os parâmetros do código sem trazer mais informações sobre os testes conclusivos finais. Porém pude perceber que com o uso do Jest permite testar funções isoladamente mockando dependências como bancos de dados e APIs.

Verifica comportamentos com toHaveBeenCalledWith, toThrow e mockResolvedValue.

Garante confiabilidade, evita regressões e documenta o código de forma automatizada.

Link do Github: https://github.com/Ualisongithub/TesteSonar02