Testes de Backend

Testes de backend são essenciais para garantir que os serviços e APIs funcionem corretamente e de forma eficiente. Eles ajudam a identificar problemas antes que o software seja implantado em produção, garantindo a confiabilidade e a qualidade do sistema. Nesta apostila, abordaremos como pensar em cenários de teste, organizar suítes de teste e validar corretamente os resultados, com foco no uso de Mocha e Chai para testes em Node.js.

1. Pensando em Cenários de Testes

1.1. Identificação dos Cenários

Os cenários de teste devem ser baseados nos requisitos funcionais e não funcionais do sistema. Para uma API, por exemplo, você pode começar identificando:

- **Cenários de Sucesso:** Situações em que a API deve retornar respostas corretas.
- **Cenários de Falha:** Situações em que a API deve lidar com erros de forma graciosa.
- **Cenários de Limite:** Testar com valores nos limites dos dados aceitos para ver como o sistema se comporta.
- **Cenários de Exceção:** Testar como o sistema lida com entradas inválidas ou situações inesperadas.

1.2. Exemplos de Cenários

Para uma API REST que gerencia usuários, alguns cenários podem incluir:

1. **Criar Usuário:** Verificar se a criação de um novo usuário funciona corretamente.
2. **Criar Usuário com Dados Inválidos:** Verificar se o sistema retorna um erro apropriado ao tentar criar um usuário com dados inválidos.
3. **Obter Usuário:** Verificar se a API retorna corretamente os detalhes de um usuário existente.
4. **Obter Usuário Não Existente:** Verificar se a API retorna um erro ao tentar obter um usuário que não existe.
5. **Atualizar Usuário:** Verificar se a atualização de dados de um usuário existente é feita corretamente.
6. **Excluir Usuário:** Verificar se a exclusão de um usuário funciona corretamente.

2. Organização das Suítes de Testes

2.1. Estrutura das Suítes

Organize seus testes em suítes (ou blocos) para facilitar a leitura e manutenção. Com Mocha, você pode usar **describe** para definir uma suíte de testes e **it** para definir casos de teste individuais.

```
const chai = require('chai');
const chaiHttp = require('chai-http');
const app = require('../app'); // Sua aplicação
const { expect } = chai;

chai.use(chaiHttp);

describe('User API', () => {

    describe('POST /users', () => {
        it('should create a new user', (done) => {
            chai.request(app)
                .post('/users')
                .send({ username: 'testuser', email: 'test@example.com' })
                .end((err, res) => {
                    expect(res).to.have.status(201);
                    expect(res.body).to.have.property('id');
                    done();
                });
        });

        it('should return an error for missing fields', (done) => {
            chai.request(app)
                .post('/users')
                .send({ email: 'test@example.com' }) // Missing username
                .end((err, res) => {
                    expect(res).to.have.status(400);
                    expect(res.body).to.have.property('error');
                    done();
                });
        });
    });

    describe('GET /users/:id', () => {
        it('should get a user by id', (done) => {
            chai.request(app)
                .get('/users/1')
                .end((err, res) => {
                    expect(res).to.have.status(200);
                    expect(res.body).to.have.property('username');
                    done();
                });
        });

        it('should return an error for non-existent user', (done) => {
            chai.request(app)
                .get('/users/9999')
                .end((err, res) => {
                    expect(res).to.have.status(404);
                    expect(res.body).to.have.property('error');
                    done();
                });
        });
    });

    // Mais testes...
});
```

2.3. Categorias de Testes

- **Testes de Unidade:** Testam componentes individuais, como funções ou métodos isolados.
- **Testes de Integração:** Testam a interação entre diferentes componentes ou serviços.
- **Testes Funcionais:** Verifica se a aplicação atende aos requisitos funcionais.
- **Testes de Aceitação:** Avaliam se o sistema atende aos critérios de aceitação definidos pelo cliente ou equipe de produto.
- Testes de API: Verifica mensagens e código de retorno dos endpoints da API.

Pense no tipo de teste mais adequado para seu projeto. Escolha entre testes de unidade ou testes de API (rotas).

3. Validações Importantes

3.1. Status Codes

Verifique se o endpoint retorna os códigos de status HTTP corretos, como 200 para sucesso, 400 para solicitações inválidas, 404 para recursos não encontrados e 500 para erros do servidor.

3.2. Estrutura da Resposta

Verifique a estrutura da resposta, como a presença de campos obrigatórios e o formato dos dados. Por exemplo, se uma resposta JSON deve incluir id, username, e email.

```
expect(res.body).to.have.property('id');
expect(res.body).to.have.property('username');
expect(res.body).to.have.property('email');
```

3.3. Validação de Dados

- **Tipos de Dados:** Verifique se os tipos de dados retornados são os esperados (string, número, etc.).
- **Valores:** Verifique se os valores retornados estão corretos, especialmente em cálculos ou manipulações de dados.
- **Campos Obrigatórios:** Certifique-se de que campos obrigatórios estejam presentes na resposta.
- **Restrições de Negócio:** Verifique se as regras de negócio estão sendo aplicadas corretamente, como limite de tamanho de campos ou formato de dados.

3.4. Tempo de Resposta

Especialmente em APIs, o tempo de resposta pode ser importante. Use ferramentas para verificar se a resposta é retornada dentro de um tempo aceitável. Se notar alguma demora em algum endpoint, investigue-o, quem sabe seja um caso de paginação (um exemplo de funcionalidade avançada que discutimos em sala de aula).

4. Práticas Adicionais

4.1. Mocking

Ao testar funcionalidades que dependem de serviços externos (como bancos de dados ou APIs de terceiros), use bibliotecas para criar mocks e stubs, evitando a necessidade de chamar os serviços reais.

4.2. Teste em Ambiente de Desenvolvimento

Execute os testes em um ambiente de desenvolvimento ou em um ambiente de teste isolado para evitar interferência com dados ou sistemas em produção.

4.3. Automação de Testes

Automatize a execução dos testes com ferramentas de CI/CD (como Jenkins, GitLab CI, ou GitHub Actions) para garantir que eles sejam executados em cada commit ou antes de cada implantação.

EXERCÍCIO PRÁTICO PARA ENTREGA 02:

Elabore um documento que **descreva os cenários de teste** que você irá desenvolver. Comece a trabalhar nele detalhando a quantidade de testes realizados, os cenários abrangidos e as suítes de testes planejadas. A formatação é flexível e não precisa seguir um modelo específico de documentação de testes. O foco deve ser em fornecer uma explicação clara e compreensível sobre quais cenários foram testados e como foram abordados.