

Testes de backend

José Glauber UFCG 2025.1

Contexto geral

É impossível garantir que um software funcione sem erros, isso se dá pela complexidade dos processos envolvidos.

Falhas podem surgir por inúmeros motivos:

- ↳ especificação incompleta ou errada
- ↳ requisitos impossíveis de serem implementados
- ↳ implementação errada

Contexto geral

O processo de desenvolvimento de testes de software é visto como uma parcela do processo de **qualidade de software**.

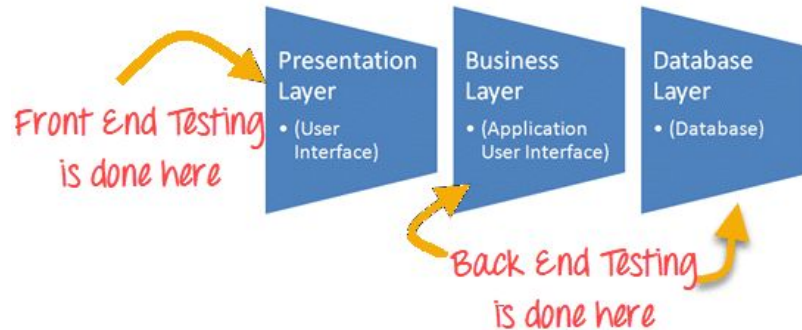
Atributos qualitativos:

1. Funcionalidade
2. Confiabilidade
3. Usabilidade
4. Eficiência
5. Manutenibilidade
6. Portabilidade

O que são testes de software?

Um das formas de verificar se a aplicação opera com qualidade, se relacionando com o conceito de **verificação** e **validação**.

Utilizar o produto desenvolvido e encontrar defeitos.



A importância dos testes em aplicações backend

Testar apenas frontend é mais fácil pois existe uma interação com a interface gráfica, o que torna mais intuitivo.

Qual o problema de só testar com interações do frontend?

1. Redução da capacidade de investigação;
2. Impossibilidade de garantir confiabilidade do backend;

identificar e corrigir bugs e falhas antes que eles se tornem problemas maiores!!!

Testes manuais X automatizados

Com o aumento da complexidade dos softwares atuais, aumenta também a chance de erros passarem despercebidos em versão de produção.

Surgem os chamados testes automatizados...



São scripts que utilizam as entradas e saídas de um software para simular um usuário ou sistema.

Mas também temos os testes manuais...



Temos a presença de testadores humanos, que irão desde a identificação do cenário de teste até a execução do passo a passo sem a utilização de ferramentas de automação.

Tipos de testes no backend



Ambiente de teste

Ferramentas e bibliotecas

Javascript



Java

JUnit

TestNG

AssertJ
Fluent assertions for java

Python



nose
is nicer testing for python



Mocha e Chai

Mocha: um framework de teste de JS, que facilita a escrita e execução de testes assíncronos. Permite diversos estilos de **assertivas**, permitindo a escolha de diferentes bibliotecas de asserção, como **Chai**.

describe

it

before

beforeEach

afterEach

Mocha e Chai

Describe: usada para agrupar testes relacionados em uma mesma suíte de teste. Fornece um bloco de escopo onde você pode definir seus casos de teste (it) e definir ganchos (before, after e etc) que afetam todos os **testes** dentro desse bloco.

It: é a função utilizada para definir um caso de testes individual.

```
describe('Função soma', function() {  
  it('deve retornar 3 quando somar 1 e 2', function() {  
    // teste aqui  
  });  
  
  describe('quando passado um número negativo', function() {  
    it('deve retornar -1 quando somar 1 e -2', function() {  
      // teste aqui  
    });  
  });  
});
```

Cada **it** representa um cenário de teste único que valida uma parte específica da funcionalidade. Quando um teste falha, o **it** fornece uma descrição clara do cenário que não funcionou como esperado

Mocha e Chai

Before e after: são funções de gancho que executam código antes ou depois de todos os testes.

Before: para preparar o ambiente.

After: limpar ou restaurar o ambiente após a execução dos testes

```
const mongoose = require('mongoose');
const { expect } = require('chai');
const User = require('./userModel');

describe('Teste do modelo User', function() {
  // Conectar ao banco de dados antes de rodar os testes
  before(async function() {
    await mongoose.connect('mongodb://localhost/testdb', {
      useNewUrlParser: true,
      useUnifiedTopology: true
    });
  });
});
```

```
// Limpar a coleção de usuários após todos os testes
after(async function() {
  await mongoose.connection.db.dropDatabase(); // Limpar o banco de dados
  await mongoose.disconnect(); // Desconectar do banco de dados
});
```

```
// Teste para verificar se um usuário é salvo corretamente
it('deve salvar um novo usuário no banco de dados', async function() {
  const user = new User({ name: 'John Doe', email: 'john@example.com' });
  const savedUser = await user.save();
});
```

Mocha e Chai

beforeEach e **afterEach**: são funções de gancho que executam código antes ou depois de cada testes.

BeforeEach: para preparar o ambiente ou estado de cada teste, garantindo que cada teste comece em condições controladas.

AfterEach: limpar o estado ou restaurar o ambiente após cada teste, para evitar interferência dos dados.

```
const { expect } = require('chai');
const TodoList = require('./todoList');

describe('TodoList', function() {
  let todoList;

  // Este bloco será executado antes de cada teste
  beforeEach(function() {
    todoList = new TodoList(); // Cria uma nova lista de tarefas antes de cada teste
  });

  // Este bloco será executado após cada teste
  afterEach(function() {
    todoList = null; // Limpa a referência à lista de tarefas após cada teste
  });

  it('deve adicionar uma tarefa à lista', function() {
    todoList.addTask('Comprar leite');
    expect(todoList.getTasks()).to.include('Comprar leite');
  });

  it('deve remover uma tarefa da lista', function() {
    todoList.addTask('Estudar JavaScript');
    todoList.removeTask('Estudar JavaScript');
    expect(todoList.getTasks()).to.not.include('Estudar JavaScript');
  });

  it('deve começar com uma lista vazia', function() {
    expect(todoList.getTasks()).to.be.empty;
  });
});
```

Mocha e Chai

Chai: é uma biblioteca de asserções que pode ser usada em conjunto com outros frameworks de teste.

assert

expect

should

Mocha e Chai

```
const soma = require('./math');  
const { expect } = require('chai'); // Importando o estilo 'expect' do Chai  
  
describe('Função soma', function() {  
  it('deve retornar 5 quando somar 2 e 3', function() {  
    expect(soma(2, 3)).to.equal(5);  
  });  
  
  it('deve retornar 0 quando somar 0 e 0', function() {  
    expect(soma(0, 0)).to.equal(0);  
  });  
});
```

Mocha e Chai

```
const assert = require('assert');
const add = require('./math');

describe('add', function() {
  it('deve retornar 5 quando somamos 2 e 3', function() {
    const result = add(2, 3);
    assert.strictEqual(result, 5);
  });

  it('deve retornar 0 quando somamos -1 e 1', function() {
    const result = add(-1, 1);
    assert.strictEqual(result, 0);
  });

  it('deve retornar -5 quando somamos -2 e -3', function() {
    const result = add(-2, -3);
    assert.strictEqual(result, -5);
  });
});
```

```
const should = require('chai').should(); // Importa e inicializa
const add = require('./math');

describe('add', function() {
  it('deve retornar 5 quando somamos 2 e 3', function() {
    const result = add(2, 3);
    result.should.equal(5);
  });

  it('deve retornar 0 quando somamos -1 e 1', function() {
    const result = add(-1, 1);
    result.should.equal(0);
  });

  it('deve retornar -5 quando somamos -2 e -3', function() {
    const result = add(-2, -3);
    result.should.equal(-5);
  });
});
```


Exercício

Temos a classe calculadora que possui os métodos **adição**, **subtração**, **multiplicação** e **divisão**.

Quais são os cenários de testes que podem surgir dessa implementação?

Exercício

Temos a classe `StringUtils` que possui os métodos `maiúsculas`, `minúsculas`, `capitalizar` e `reverter`.

Quais são os cenários de testes que podem surgir dessa implementação?

Testes unitários

O que são testes unitários?

Verifica o funcionamento de unidades individuais. Isola a unidade a ser testada e fornece entradas para verificação correta da saída.

Testes unitários permitem a criação de uma suíte de **testes automatizados**, que pode ser executada repetidamente para verificar a integridade da aplicação.

Deteção de erros no estágio inicial do desenvolvimento.

Como funciona os testes unitários?

São executados sempre que o código é modificado, testando funções, métodos ou classes.

```
function soma(a, b) {  
  return a + b;  
}  
  
module.exports = soma;  
  
const chai = require('chai');  
const expect = chai.expect;  
const soma = require('./math');  
  
describe('Função soma', function() {  
  it('deve retornar 3 quando os parâmetros são 1 e 2', function() {  
    expect(soma(1, 2)).to.equal(3);  
  });  
  
  it('deve retornar -2 quando os parâmetros são -1 e -1', function() {  
    expect(soma(-1, -1)).to.equal(-2);  
  });  
  
  it('deve retornar 0 quando os parâmetros são 0 e 0', function() {  
    expect(soma(0, 0)).to.equal(0);  
  });  
});
```

> mocha

Função soma

- ✓ deve retornar 3 quando os parâmetros são 1 e 2
- ✓ deve retornar -2 quando os parâmetros são -1 e -1
- ✓ deve retornar 0 quando os parâmetros são 0 e 0

3 passing (XXms)

Quais os benefícios?

- Melhoria da qualidade do código
- Maior confiança na modificação
- Aceleração do processo de desenvolvimento
- Redução dos custos
- Melhoria da colaboração

Boas práticas

Atomicidade: Escrever testes que se concentram em uma única unidade de código ou aspecto específico da funcionalidade.

Manutenção!!!!

Boas práticas

Independência: o resultado de um teste não deve depender do outro. Cada teste deve ser capaz de ser executado de forma isolada.

Isolamento de problemas!!!

Boas práticas

Nomeação clara: os testes devem ter nomes claros e descritivos que expliquem exatamente o que está sendo testado e qual o comportamento esperado.

Compreensão rápida!!!

```
// Nomeação clara
it('deve retornar 5 quando 2 e 3 são somados', function() {
  expect(soma(2, 3)).to.equal(5);
});

it('deve retornar 0 quando 0 e 0 são somados', function() {
  expect(soma(0, 0)).to.equal(0);
});

// Nomeação não clara
it('deve funcionar corretamente', function() {
  expect(soma(2, 3)).to.equal(5);
});
```

Escrita de testes - Mocking

Permite simular o comportamento de objetos complexos ou externos, facilitando a verificação da lógica interna do código sob teste.

Mock



Isolamento de testes



Testes mais rápidos







Testes com cenários difícil de simular

Testes de API

O que são testes de API?

Envia solicitações a uma API e verificar suas respostas. Isso inclui verificar se a API retorna os dados esperados, lida corretamente com erros e se atende aos requisitos de **desempenho** e **segurança**.

- Testes funcionais  Verificação de endpoints e validação das respostas
- Testes de validação de dados  Consistência dos dados
- Testes de comportamento de erros  Tratamento de erros e códigos de status HTTP
- Testes de performance  Tempo de resposta e de teste de carga

Escrita de testes

```
const axios = require('axios');
const { expect } = require('chai');

describe('API Testes', function () {
  it('Deve retornar 200 para a rota GET /users', async function () {
    const response = await axios.get('https://api.exemplo.com/users');
    expect(response.status).to.equal(200);
  });

  it('Deve criar um novo usuário e retornar 201', async function () {
    const response = await axios.post('https://api.exemplo.com/users', {
      name: 'João',
      email: 'joao@example.com'
    });
    expect(response.status).to.equal(201);
    expect(response.data).to.have.property('id');
  });
});
```

Exercício

Criem um arquivo de teste que reflete um cenário para uma das rotas do seu projeto.

```
describe('Testes da API', function () {

  it('Deve retornar 200 para a rota GET /users', function (done) {
    request(app)
      .get('/users')
      .expect(200)
      .expect('Content-Type', /json/)
      .end((err, res) => {
        if (err) return done(err);
        expect(res.body).to.be.an('array');
        expect(res.body.length).to.be.gte(2);
        done();
      });
  });

  it('Deve criar um novo usuário e retornar 201', function (done) {
    request(app)
      .post('/users')
      .send({
        name: 'Charlie',
        email: 'charlie@example.com'
      })
      .expect(201)
      .expect('Content-Type', /json/)
      .end((err, res) => {
        if (err) return done(err);
        expect(res.body).to.have.property('id');
        expect(res.body.name).to.equal('Charlie');
        expect(res.body.email).to.equal('charlie@example.com');
        done();
      });
  });
});
```

Melhores práticas e cobertura

O que é cobertura de código?

Mede a quantidade de código que é executada quando os testes são executados.

A cobertura de código ajuda a identificar áreas do código que podem **precisar de mais testes** e a garantir que a maior parte possível do código está sendo testada.

Instabul (nyc) com mocha.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	98.92	94.36	99.49	100	
yargs	99.17	93.95	100	100	
index.js	100	100	100	100	
yargs.js	99.15	93.86	100	100	
yargs/lib	98.7	94.72	99.07	100	
command.js	99.1	98.51	100	100	
completion.js	100	95.83	100	100	
obj-filter.js	87.5	83.33	66.67	100	
usage.js	97.89	92.59	100	100	
validation.js	100	95.56	100	100	

Melhores práticas

Testar todos os casos de uso

- Testes de casos normais e extremos

Escrever testes claros e manuteníveis

- Nomenclatura clara

Utilizar testes unitários e de API

- Simular funções individuais como também verificar as respostas das requisições.

Melhores práticas

Usar mocking de forma adequada

- Isolar unidades
- Manter simplicidade

Escrever testes repetíveis

Documentar testes

- comentários, boas descrições e README

Referências

- "Testing JavaScript Applications" - Lucas Fernandes
- "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" - Jez Humble & David Farley
- "The Art of Software Testing" - Glenford J. Myers
- "A Comprehensive Guide to Unit Testing in Node.js" - FreeCodeCamp
- "End-to-End Testing for Modern Web Applications" - Martin Fowler
- "Testing Best Practices for Node.js Applications" - Rising Ode
- "Node.js Testing" - Node.js Documentation