Tópicos Especiais em Desenvolvimento Back End

Desenvolvimento Node.js

Lucas Goulart Silva

2021

Tópicos Especiais em Desenvolvimento Back End

Desenvolvimento Node.js

Lucas Goulart Silva

Sumário

Capítulo 1. Testes Unitários

Testes de software são uma importante etapa integrante do processo de desenvolvimento de software. Por meio dos testes, é possível identificar cenários que necessitem de ajustes ou correções antes que esses erros aconteçam de fato. Neste capítulo, serão abordados os seguintes conceitos:

- Testes de Software.
  - Pirâmide de Testes.
- Metodologia XP.
  - TDD.
- Testes Unitários.
- A biblioteca Jest.

1.1. Testes de Software

O surgimento do termo *"bug"* – palavra da língua inglesa que significa inseto – nos remete à ideia de um defeito ou falha que pode ser encontrada no código fonte de um programa e que, portanto, é responsável pelo mau funcionamento dele. O termo, porém, é bem mais antigo. Thomas Edison, importante empresário que financiou o desenvolvimento de diversos equipamentos eletroeletrônicos industriais, já usava o termo para se referir a problemas causados pela presença de insetos em suas ferramentas. O termo foi, posteriormente, utilizado para relatar a presença de uma traça no computador Mark II, em 1947. Desde então, o termo tem se tornado cada vez mais comum em meio aos profissionais de TI e, mais recentemente, tem sido utilizado até mesmo por pessoas cujas atividades profissionais não tenham qualquer relação com o desenvolvimento e com a engenharia de software.

Tão antigo quanto o termo é a necessidade que os envolvidos na engenharia de um produto têm em encontrá-los e corrigi-los com a maior eficiência possível. Muito além da capacidade de antever problemas por meio de um desenho – seja com cálculo, projetos, simulação ou observação –, bugs acontecem naturalmente e,

portanto, é extremamente necessário que sejam realizados testes. A submissão de produtos ao processo de testes tem por **objetivo demonstrar a presença de erros** e, assim, permitir que sejam corrigidos.

Os testes de software podem ser classificados de diferentes maneiras:

- Quando classificados com relação à característica do produto que está sendo avaliada, subdividimos os testes em testes de segurança, testes de performance – realizando-se testes com grandes volumes de dados, por exemplo; ou testes funcionais – que visam verificar se o comportamento obtido é, de fato, o adequado.
- Quando relacionados ao ponto de vista daqueles que executam os testes, estes podem ser classificados como: testes de caixa branca, quando aquele que o realiza tem acesso ao código fonte do objeto testado; ou como testes de caixa preta, quando não há acesso ao código fonte. Para este último caso, o foco dos testes é exclusivo no comportamento e nos resultados esperados.

Como toda interação humana é sujeita a falhas, as boas práticas recomendam que os testes devam ocorrer de forma automatizada. Testes devem, portanto, ser codificados por meio de uma linguagem de programação, em algoritmos que descrevam o comportamento e os resultados esperados. Desta forma, é possível garantir seu maior sucesso.

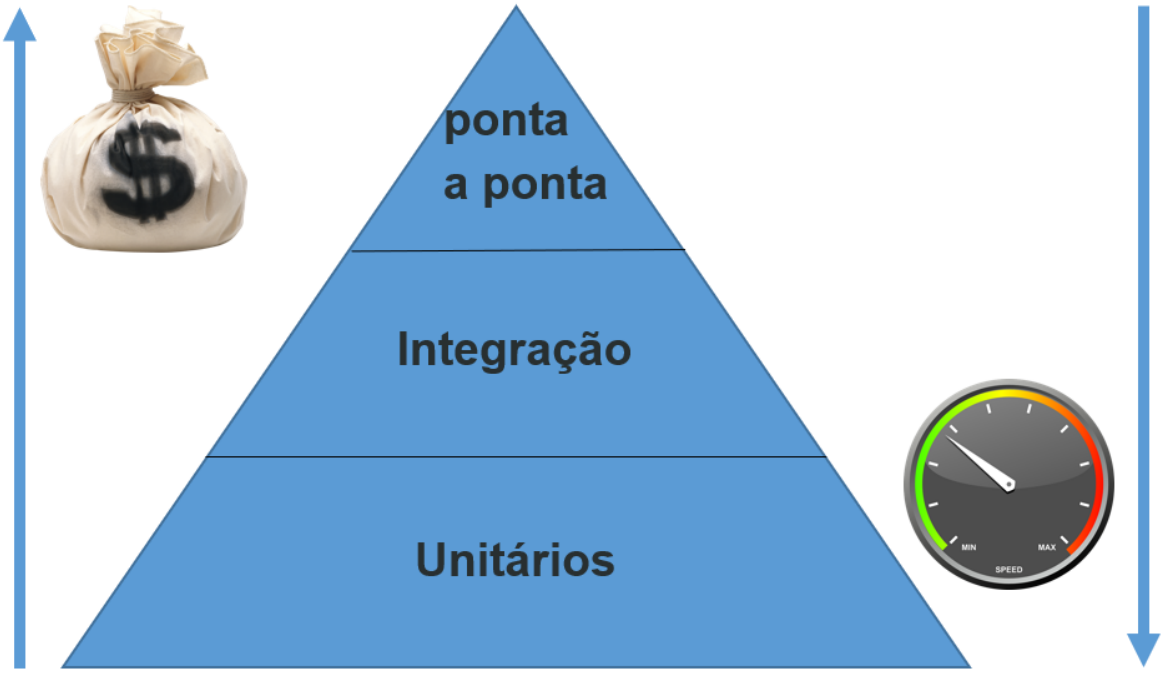
Nos modelos mais tradicionais, existe a ideia de que os testes sejam realizados durante uma fase específica do ciclo de vida de desenvolvimento. Contudo, é um consenso cada vez mais comum que é de fundamental importância, para a qualidade e garantia de sucesso de um projeto, que ocorram etapas de verificação durante todo o ciclo de desenvolvimento. Dessa maneira, é possível encontrar falhas antecipadamente e corrigi-las o quanto antes.

Com relação à etapa do desenvolvimento em que ocorrem, é possível classificar os testes em testes de unidade, testes de integração e testes de ponta a ponta (ou aceitação).

A visualização dessa classificação, por meio da pirâmide de testes, permite discorrer sobre as principais características dos itens presentes em cada uma destas etapas:

- É na **base** que estão presentes os **testes unitários**, pois são os primeiros a serem construídos. A maior largura indica que o volume de testes unitários será maior do que os demais testes;
- É no **meio** que são apresentados os testes de integração, cuja principal finalidade é integrar pequenas unidades e verificar o seu comportamento quando atuando em conjunto;
- No **topo**, os testes de ponta a ponta representam aqueles com marcos de maior importância no projeto e que, normalmente, estão associados a entregas e finais de ciclo.

Figura 1 - Pirâmide de Testes



Os testes tendem a se tornar mais rápidos na medida em que se caminha do topo para a base da pirâmide. Por outro lado, os testes tendem a se tornar mais caros e valiosos na medida em que se caminha para o topo da pirâmide.

Um resumo sobre as principais características dos testes, de acordo com a etapa em que ocorrem, pode ser visualizado na **Tabela 1 - Características dos Testes por Etapa**.

Tabela 1 - Características dos Testes por Etapa

	Unitários	Integração	Ponta a ponta
Objetivo	Testar uma pequena unidade – pode ser uma função ou classe	Testar a integração entre algumas unidades do sistema	Testar o sistema como um todo
Periodicidade	Constantemente durante o desenvolvimento	Diariamente durante o desenvolvimento	Ao final de uma etapa
Quantidade	Grande	Intermediária	Pequena
Velocidade	Rápido	Razoável	Lento
Valor agregado	Pouco	Mediano	Muito
Custo	Baixo	Intermediário	Alto

1.2. Metodologia XP

A programação extrema (e***X***treme ***P***rogramming – XP) é uma metodologia de desenvolvimento de software focada na geração de entregas de qualidade. É considerada uma metodologia de desenvolvimento de software ágil e, portanto, prevê a capacidade de que as entregas ocorram em pequenos ciclos, em contrapartida ao tradicional modelo de cascata.

Em meio às revoluções tecnológicas acarretadas pela rápida expansão da internet em meados dos anos 90, um desenvolvedor de software chamado Kent Beck

percebeu que a realização de entregas de qualidade não era suportada pelos processos da época. Ele, então, decidiu agrupar uma séria de boas práticas, algumas das quais já existentes à época, mas ainda utilizadas de forma isolada, criando, assim, uma metodologia inovadora.

Uma das principais ideias da metodologia XP é que boas práticas devem ser levadas ao extremo para garantir a boa qualidade das entregas. São algumas das boas práticas preconizadas:

- *Pair-programming*: Também chamada de programação em pares, é uma técnica que prega que dois programadores trabalhem juntos em um único ambiente. Enquanto o primeiro escreve o código, o outro observa o resultado que está sendo produzido. Ao final do processo, os envolvidos devem negociar a melhor solução para o problema que estão resolvendo;
- *Stand-up meeting*: Realização de reuniões diárias nas quais os participantes devem permanecer de pé, evitando, assim, que durem mais tempo do que o considerado estritamente necessário;
- **Testes unitários:** Testes unitários devem ser desenvolvidos constantemente e têm por objetivo encontrar falhas em cada um dos componentes do sistema produzido. A forma pela qual os testes são produzidos é representada pelos ciclos do TDD.

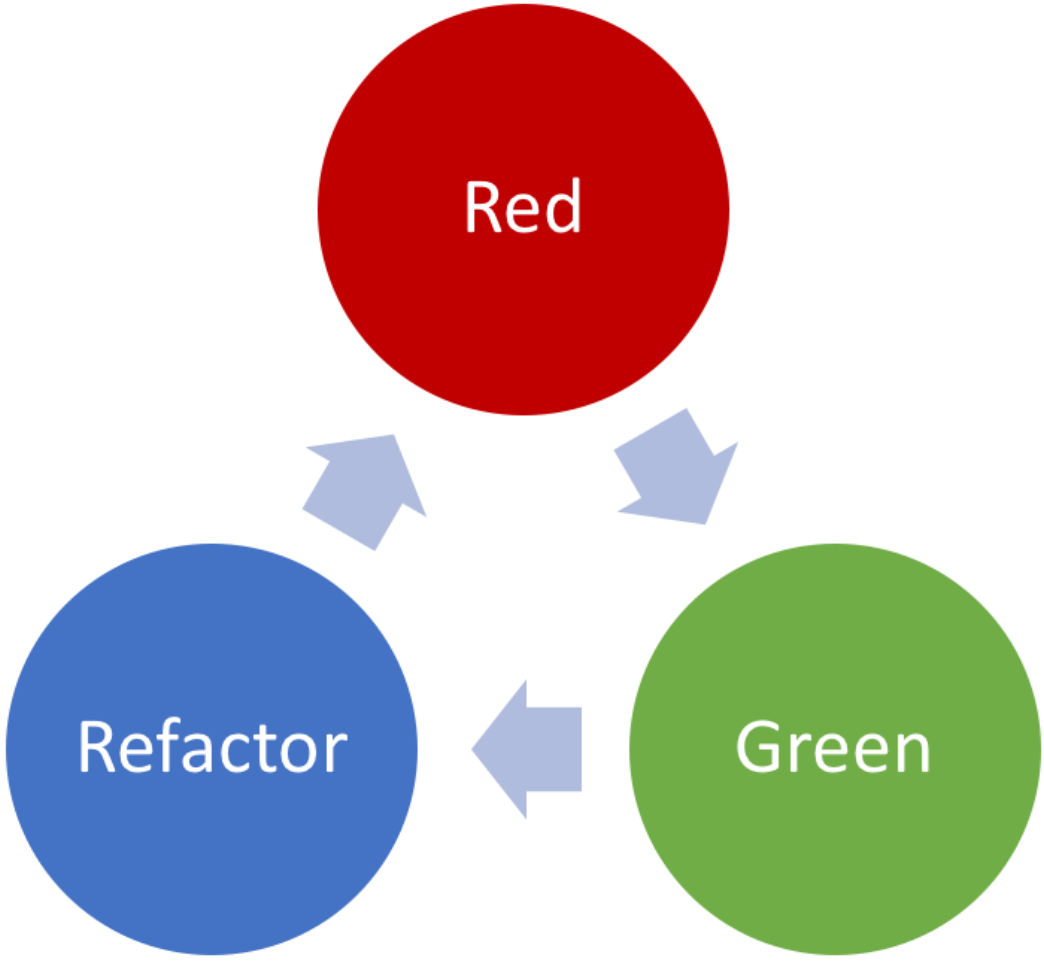
O TDD é uma técnica **de desenvolvimento de software** onde os testes devem guiar o desenvolvedor na criação das tarefas. Os testes devem ser, portanto, criados antes do código de produção. Inicialmente, a ideia surgiu como uma necessidade para projetos ágeis, especialmente o XP. Mas, com o passar do tempo, essa técnica foi ganhando cada vez mais espaço, sendo amplamente adotada na indústria do desenvolvimento de software. Hoje em dia, o TDD é amplamente utilizado mesmo para casos em que se esteja trabalhando com metodologias mais tradicionais.

A técnica prevê a existência de 3 fases e, para cada uma das fases, o desenvolvedor deve desempenhar um papel distinto. Ao final da terceira e última fase, o processo se inicia novamente e novos recursos serão criados.

FASE 01 – TESTE

O desenvolvedor deve codificar um novo teste. O teste deve, então, ser executado. Como a função que atende a esse código ainda não existe, o teste deve falhar. Caso não ocorram falhas, o ciclo se encerra neste ponto e nenhuma alteração será realizada no código. Desta forma, fica claro que a motivação para a codificação é a existência de um teste que esteja falhando.

Figura 2 - Ciclos do TDD



FASE 02 – CODIFICAÇÃO

Após visualizar um teste falhando, o foco do desenvolvedor passa, então, a ser a escrita de uma solução que possa atender ao novo teste que está falhando. Ao mesmo tempo, o desenvolvedor deve se manter atento para que suas alterações não

quebrem os outros testes. Nesta fase não é esperado que o código desenvolvido apresente a solução ótima. Pequenas imperfeições são aceitáveis a fim de garantir a entrega do teste passado.

FASE 03 – REFATORAÇÃO

O ciclo de refatoração é aquele no qual o desenvolvedor poderá alterar a forma como a solução está implementada sem, no entanto, alterar o que está sendo feito. Como resultado deste ciclo, é esperado um código mais limpo, uma solução menos acoplada, sem, no entanto, alterar o que a solução de fato faz.

As consecutivas execuções dos ciclos são chamadas de *baby steps*, porque remetem ao processo cuidadoso de evoluir a aplicação em pequenos passos, semelhante a passos de bebê.

1.3. Testes Unitários

Testes unitários são aqueles criados para verificar o correto funcionamento de uma unidade de código. Normalmente, essas unidades são representadas por uma classe, uma função, ou um pequeno conjunto de funções que possam ser semanticamente agrupadas. São testes de caixa branca, automatizados e normalmente tem caráter de atender a uma especificação funcional.

A fim de isolar cada uma das unidades testadas durante o desenvolvimento de testes unitários, é bastante comum que sejam utilizados simuladores para outras unidades senão a que está sendo submetida aos testes. Esses simuladores são chamados de *Mock* e têm a função de simular de maneira simples, por vezes até mesmo rudimentar, o comportamento de outros módulos do sistema. *Mocks* são de fundamental importância para que se possa realizar testes unitários em unidades que possuem um conjunto grande de interfaces com outras unidades.

Os testes unitários são localizados na base da pirâmide e são, portanto, os primeiros a serem desenvolvidos e, normalmente, os mais volumosos do conjunto de testes automatizados de uma aplicação. Por realizarem testes em unidades isoladas do código, os testes unitários não são capazes de agregar grande valor à entrega,

uma vez que não asseguram o correto funcionamento das unidades em conjunto. Porém, devido a sua velocidade de realização e baixo custo, são de grande importância.

1.4. Jest

Jest é um framework para testes em Javascript que suporta diversas das bibliotecas mais utilizadas do mercado, como React, Angular e **Node.js**. A arquitetura do framework é dividida em dois pontos principais: um executor de testes (*test-runner*) responsável por localizar os arquivos de teste do projeto e verificar sua execução; uma biblioteca de comandos que permite que os desejos sejam expressados na forma de assertivas, que o framework dá o nome de **matchers.**

O Jest possui matchers para comparação de números, textos, booleanos, arrays, entre outros.

Figura 3 - Exemplo de teste unitário escrito com Jest

```
test('Valor da soma das prestações deve ser igual ao montante com duas casas decimais', () => {
  // Dado (given)
  const numeroPrestacoes = 3
  const montante = 100

  // Quando (when)
  const prestacoes = calculaValor.calcularPrestacoes(montante, numeroPrestacoes)

  // Então (then)
  expect(prestacoes.length).toBe(numeroPrestacoes)
  expect(prestacoes).tenhaSomaDeValoresIgual(montante)
  expect(prestacoes).sejaDecrescente()
})
```

No exemplo apresentado na Figura 3 - Exemplo de teste unitário escrito com Jest, é possível visualizar algumas das funções mais básicas do Jest:

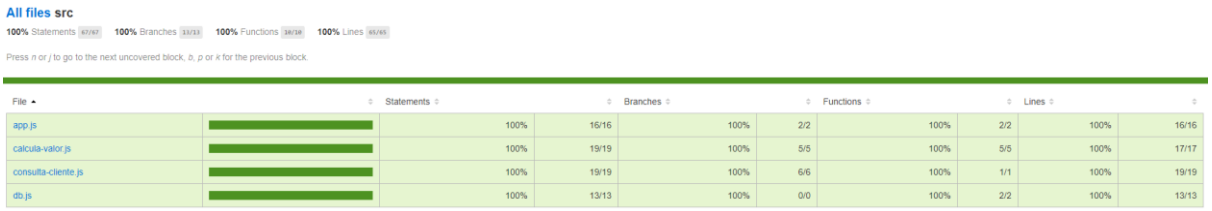
- test: Função que cria um teste. Deve receber, como parâmetros, uma *string* com o nome do teste e uma função que executa o teste;
- expect: Função utilizada após a realização de um teste contendo uma expectativa de resultado que deve ser atendida pelo teste. Recebe, como parâmetro, o valor real;

- toBe: Uma das principais *assertions* do jest é utilizada encadeada com a função ‘expect’. Caso os valores esperado e real sejam iguais, o teste é considerado sucesso. Caso contrário, o teste falha.

Dada uma aplicação em Javascript, a sua instalação pode ser feita por meio do comando **‘npm install –save-dev jest’**. A configuração inicial poderá ser feita por meio do comando **‘jest –init’**. Após realizada a configuração, a realização dos testes deverá ocorrer por meio do comando **‘npm test’**.

O módulo de execução de testes do Jest é capaz de coletar estatísticas dos testes realizados e gerar um relatório contendo as principais métricas.

Figura 4 - Relatório de Cobertura do Jest



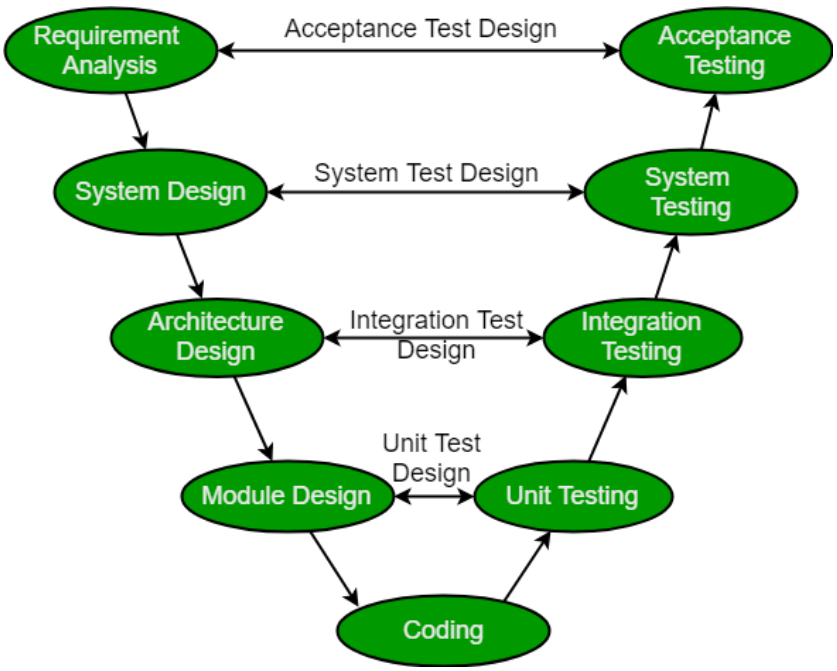
Capítulo 2. Testes de Integração

Testes de integração são previstos pela metodologia XP como de fundamental importância para garantir a qualidade do software. Estes devem ser executados diariamente, usando, como base, o código já integrado ao Sistema de Controle de Versão. São criados para verificar o correto funcionamento das unidades quando executadas de forma integrada. Normalmente, são capazes de representar uma visão bem próxima das regras de negócio solicitadas pelo cliente.

Normalmente, durante a realização dos testes de integração é o momento em que ocorre a primeira interação entre componentes e, portanto, importantes módulos da aplicação que, até então, estavam isolados, passam a ser requisitados. Por exemplo: Banco de Dados, Cache, entre outros. Por isso, podem fornecer métricas a respeito de itens não funcionais, como o desempenho da aplicação em determinados cenários.

De acordo com o modelo em V do Ciclo de Vida de Desenvolvimento de Software, os Testes de Integração estão para o desenho arquitetural da aplicação, assim como os Testes Unitários estão para o desenho de um módulo.

Figura 5 - Modelo em V do Ciclo de Vida de Desenvolvimento de Software



2.1. supertest

O **supertest** é uma biblioteca do Node.js para a realização de testes de WEB APIs utilizando o Jest. São, portanto, muito utilizados para o desenvolvimento de testes de integração.

Dada uma aplicação em Javascript que esteja expondo uma interface REST e que já tenha o **express** instalado, a instalação do supertest pode ser feita por meio do comando **'npm install –save-dev supertest'**. Para o correto funcionamento, a aplicação deve ter os arquivos de inicialização e montagem dos endpoints separados para evitar que a porta HTTP, utilizada pela aplicação, fique em uso após a execução dos testes.

Figura 6 - app.js

```
const express = require('express')
const app = express()

app.use(express.json())

app.get('/', async (req, res) => {
  res.status(200).send('Bootcamp desenvolvedor back end - Tópicos especiais!')
})

module.exports = app
```

Figura 7 - server.js

```
const app = require('./app')

const db = require('./db')

app.listen(5678, () => {
  console.log('Aplicação de exemplo ouvindo na porta 5678!')
})
```

Figura 8 - server.test.js

```
const supertest = require('supertest')

const request = supertest('http://localhost:5678')

test('Servidor na porta 5678', async () => {
  const resposta = await request.get('/')
  expect(resposta.status).toBe(200)
})
```

O método de teste apresentado na **Figura 8 - server.test.js** apresenta uma única mudança em relação ao previamente observado nos testes unitários. Por se tratar de um teste que faz uma requisição, a função é declarada como assíncrona, utilizando o modelo **async/await**. Também seria possível utilizar uma abordagem com uma **promise**.

A implementação dos métodos de teste de integração utilizando o Jest segue a mesma lógica semântica da utilizada nos testes unitários. A maior diferença é dada pelo ponto de vista dos objetivos de cada teste.

2.2. Configuração e Desmontagem

Quando se está trabalhando com um framework de testes automatizados, é bem comum que o desenvolvedor precise executar cenários de configuração e desmontagem dos testes. Isso é requerido especialmente por dois motivos:

- Caso exista algum recurso a ser alocado e que viabilize o cenário desejado. Por exemplo, estabelecendo uma nova conexão com um banco de dados.
- Caso exista algum processamento necessário para iniciar os testes e que monte o cenário desejado. Por exemplo, inserindo dados no banco.

É interessante observar que, normalmente, quando é necessário fazer a montagem, quase sempre também será necessário realizar a desmontagem. O que

nos exemplos acima significaria remover os dados de teste e desconectar do banco. Observe que realizar a montagem dentro do caso de teste, apesar de ser, por vezes, inviável, é bastante possível. Por outro lado, realizar a desmontagem é normalmente mais trabalhoso. Isso porque caso ocorra alguma falha dentro de um dos testes, ele será abortado e a execução do método não chegará até o fim. Desta maneira, um teste que falhasse poderia implicar em um recurso não desalocado corretamente, inviabilizando o ambiente.

As funções de configuração e desmontagem do Jest são:

- Configuração Geral – Utilizados em situações em que o processamento será utilizado para cada um dos testes.
  - beforeEach – Executado antes de cada um dos testes;
  - afterEach – Executado após cada um dos testes.
- Configuração Única – Utilizados em situações em que o processamento único é capaz de gerar a configuração para todos os testes.
  - beforeAll – Executado antes do primeiro teste;
  - afterAll – Executado após o último teste.

Figura 9 - Configuração e Desmontagem

```
beforeEach(async () =>
  await db.cliente.destroy({ where: {} })
  await db.consulta.destroy({ where: {} })
)

afterAll(async () => await db.sequelize.close())
```

A **Figura 9 - Configuração e Desmontagem** apresenta um caso em que duas das funções de montagem/desmontagem funções foram utilizadas com finalidades distintas.

A função **beforeEach** chama o método **destroy** da biblioteca **sequelize** para duas entidades: cliente e consulta. Como a propriedade **where** do objeto enviado por

parâmetro está vazia, todos os registros dessas entidades serão removidos. Dessa forma, é possível observar que o desenvolvedor deseja garantir que todos os testes serão inicializados sem nenhum registro das entidades cliente e consulta no banco de dados.

Já a função **afterAll** é chamada para que a conexão com o banco de dados seja desfeita ao final da bateria de testes e que, portanto, o recurso utilizado não esteja mais sendo utilizado.

Capítulo 3. Sistemas de Controle de Versão

<texto>

<título>

<texto>

Capítulo 4. Contêineres

<texto>

<título>

<texto>

Capítulo 5. Integração Contínua e Entrega Contínua

<texto>

<título>

<texto>

Referências

BECK, Kent. **TDD Desenvolvimento Guiado por Testes**. Bookman, 16 de junho de 2010

BECK, Kent. **Programação Extrema (XP) Explicada**: Acolha As Mudanças. Bookman, 26 de fevereiro de 2004

JEST. **npmjs**, 2021. Disponível em: https://www.npmjs.com/package/jest. Acesso em: 30 jun. 2021.

INICIANDO no Jest. JestJS, 2021. Disponível em: https://jestjs.io/pt-BR/docs/getting-started. Acesso em: 30 jun. 2021.

SUPERTEST. **npmjs**, 2021. Disponível em: https://www.npmjs.com/package/supertest. Acesso em: 30 jun. 2021.