

Uma Abordagem de Testes Assíncronos para Serviços Restful com *NODEJS* utilizando *BDD*

XXXXXX X. XXXXXXXX¹, XXXXXXXXXXXX X XXXXX²

¹XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (XXXXXXX)

XX – Brazil

²XX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.

{XXXXXXX}@xxx.xxx.br, xxxxxx@gmail.com

Abstract. *This work demonstrates how to test routes in applications RESTful of an agnostic way, automated and independent programming language in which the API was written. This work aims to demonstrate techniques BDD (Behavior Driven Development) taken in developing a social network aimed at promoting better communication between Brazilian researchers. Will be addressed BDD creation of RESTful API type CRUD using an agnostic tool called MOCHA with the framework CHAI assertion using the technology of NodeJS backend*

Resumo. *Esse trabalho demonstra como testar rotas em aplicações RESTful de uma forma agnóstica, automatizada e independente linguagem de programação em que a API foi escrita. Assim é possível demonstrar técnicas BDD (Behavior Driven Development ou desenvolvimento guiado por comportamento) praticadas no desenvolvimento de uma Rede Social, que visa promover maior facilidade de comunicação entre pesquisadores brasileiros. Será abordado o BDD criação da API RESTful do tipo CRUD, utilizando uma ferramenta agnóstica chamada de MOCHA com o framework de asserção CHAI utilizando a tecnologia de backend NODEJS.*

1. Introdução

O BDD (*Behavior Driven Development* desenvolvimento guiado por comportamento) é a técnica que deriva do *DDD(Domain Drive Design)* Desenvolvimento Guiado por Domínio, que visa buscar maior interação da equipe de desenvolvimento e os *stakeholders*. Trata-se de gerenciar o escopo do comportamento esperado em um sistema, que deve estar em sintonia com a regra de negócio introduzida e descrita pelos *stakeholders*. Logo no início do desenvolvimento do sistema, cria-se um código de teste que contempla o comportamento descrito, posteriormente este código servirá como documentação e base sólida de manutenção das fontes do sistema de forma automatizada, respondendo ao comportamento. Logo poderá satisfazer vários testes de unidade *BDD* (Test-Driven Development) (Desenvolvimento Orientado a Testes) que correspondem aos comportamentos descritos pelos *stakeholders*.

Esse conceito se aplicado de forma satisfatória terá maior consistência e facilidade de manutenção do software durante o seu desenvolvimento e vida útil. No caso particular da *API REST* o desenvolvedor e o *stakeholder* terão a certeza do comportamento de cada

verbo da rota *HTTP(Hypertext Transfer Protocol)* *GET, POST, PUT* e *DELETE*, que são o padrão de *REST* em aplicações *RESTful* do tipo *CRUD*.

Abordando assim o *BDD* de forma simples utilizando algumas tecnologias agnósticas para testes de rotas *REST* de quaisquer APIs. Quando observando a área dos softwares verifica-se o crescimento da necessidade de processos mais confiáveis na criação, o desenvolvimento e implementação, o *TDD* está em foco junto ao movimento Ágil. Contudo apenas o *TDD* não consegue criar uma base sólida em aplicações críticas como as *RESTful*, fazendo necessário o uso do *BDD* para melhorar e complementar esse conjunto.

O termo *BDD* foi cunhado por Dan North por volta e 2003(North 2009), Baraúna fala a respeito.

”Dan North estava tendo dificuldades em ensinar *TDD* para seus alunos, pois eles ficavam com dúvidas sobre por onde começar a testar, o que testar, como nomear seus testes etc”(Baraúna 2013)

A grande e visível dificuldade era ”o que testar? quando testar? e como testar?”. Essas três perguntas eram recorrentes entre os alunos, e segundo o próprio Dan North o *BDD* resolve isso de forma simples, pois trabalha no domínio do problema de forma direta, clara e declarativa diminuindo a distância entre os seus alunos e o mundo dos testes.

2. Proposta, Desenvolvimento e Metodologia

2.1. Proposta da Aplicação

Foi proposta uma aplicação na qual possui como principal objetivo criar um caminho menor entre orientadores, orientandos e pesquisadores de uma determinada instituição. Essa rede de contatos é denominada Rede de Conhecimentos do Ins*.

A Rede de Conhecimento será uma rede social focada em melhorar a comunicação dos pesquisadores brasileiros divididos em diferentes instituições de pesquisa. Essa rede visa interconectar pessoas através do conhecimento de áreas de pesquisa, projetos executados, artigos publicados e outros elementos que possam colaborar cooperativamente para a disseminação do conhecimento. A ideia por trás da Rede de Conhecimento é a ligação de pesquisadores que tenham em comum artigos, projetos e pesquisa numa mesma área. O que vai permitir a interação e a troca de conhecimento entre todos que publicam ou pesquisam determinado assunto.

O sistema que viria a ser uma rede social baseada em conhecimentos está sendo criada usando arquitetura *RESTful*, sendo usado *BDD* e usa-se a linguagem de programação *JAVASCRIPT(NODEJS)* em seu *backend*.

A aplicação baseia-se em mensagens diretas do cliente ao servidor, com o intuito de buscar em uma base de dados *noSQL* estruturada em um banco de dados de grafos, onde se concentram as informações acerca dos assuntos que estão sendo pesquisados por todos os participantes da rede, pois a Figura 01 apresentada, demonstra graficamente a modelagem desse sistema por meio de um diagrama conceitual, que aponta os relacionamentos de forma clara entre os interessados e participantes da Rede de Conhecimentos

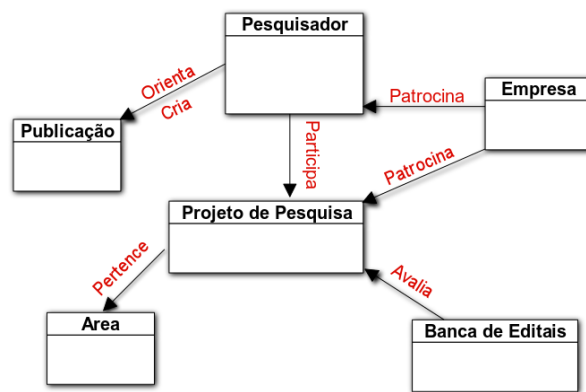


Figura 1. Diagrama de Conceitual da Rede de Conhecimentos

2.2. Desenvolvimento

Houve uma grande preocupação por parte do *stackholder* e orientador sobre a metodologia de desenvolvimento aplicada. Como o *BDD* é basicamente um espelho dos diagramas *UML* de Caso de Uso, ficou simples codificar os testes baseados nas rotas *REST*, de acordo com o comportamento descrito em tais diagramas e esperado pelo sistema.

Em sequência o desenvolvimento seguiu essa linha de criar um teste para cada comportamento esperado, de forma que segue o fundamento correto do *TDD*. Terminada a codificação do teste, inicia-se a escrita do código fonte que resolve esse caso de teste e assim por diante até o final da resolução de todos os casos testes como descrito em Aniche.

”Entretanto, uma pergunta interessante é: Será que é possível inverter, ou seja, testar primeiro e depois implementar? E mais importante, faz algum sentido?”(Aniche 2014)

2.3. Metodologias

Apesar de serem utilizados métodos ágeis para o desenvolvimento do sistema em comumhão ao *BDD*, para a pesquisa acerca dos conceitos e tecnologias usadas nesse projeto foram usado o *NODEJS* ”Node.js é uma plataforma construída sobre o motor *JavaScript* do Chrome para a construção de aplicações de rede, facilmente escaláveis rápidas.(NodeJS.org 2015)”, *MOCHA* ”Mocha é um *framework* de teste em *JavaScript* rico com recursos para execução no *NODEJS* (Mocha.org 2015)”e *CHAI*”Chai é uma biblioteca asserção *BDD / TDD* para *NODEJS* eo navegador que pode ser deliciosamente emparelhado com qualquer *framework* de testes *javascript*.”(Chai.com 2015), sendo o ponto de referência e guia para aplicar os testes.

Na aplicação que esse artigo trata, o *BDD* é usado para o desenvolvimento e documentação das rotas *RESTful*, já que *JAVASCRIPT(NODEJS)* possui *I/O(Input/Output)* assíncrono e guiado por eventos, isso pode causar problemas e muita dificuldade para a asserção de cada teste. O uso de um *Framework* organizado e documentado, no caso o *CHAI* que gerencia os testes unitários, a fazer asserções de forma declarativa foi o fator primordial nesse desenvolvimento, tendo vista que o *CHAI* é escrito em *JAVASCRIPT(NODEJS)* e pode ser usado tanto para teste do lado cliente e/ou servidor, sendo o caso em questão.

O código apresentado refere-se a códigos de testes presentes na aplicação. Nele fica o exemplo de quão descritivos e declarativos são os testes de cada rota *RESTful* da aplicação, como mostrado na listagem presente no quadro 01.

Quadro 1. Código para Testar uma Rota do Tipo Post no Sistema

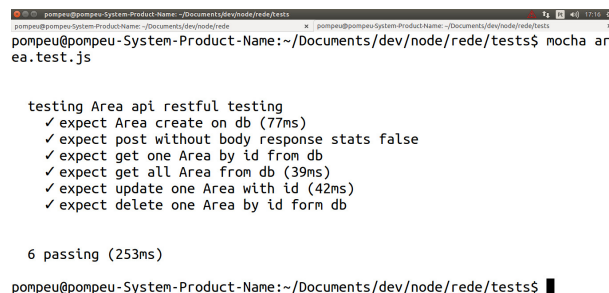
```
1 var expect = require('chai').expect,
2     superagent = require('superagent'),
3     ch = require('charlatan'),
4     url = require('url'),
5     baseUrl = 'http://localhost:3000/api/area';
6
7 describe('testing area api restful', function () {
8     var body = {
9         nomeArea : ch.Name.name(),
10        subCodigo : ch.rand(8000, min = 7001),
11        subArea : ch.Name.name(),
12        subNivel : ch.Name.title(),
13    }
14    it('expect area create on db', function (done) {
15        function endHandler(err, res) {
16            expect(err).to.not.exist;
17            expect(res).to.exist;
18            expect(res.body.status).to.true;
19            expect(res.body.err).to.null;
20            expect(res.body.result)
21                .to.an('object');
22            expect(res.body.result.id)
23                .to.an('Number');
24            expect(id).to.an('Number');
25            done();
26        }
27        superagent
28            .post(url.resolve(baseUrl, 'area'))
29            .send(body)
30            .end(endHandler);
31    });
32 });
```

Esse código apresenta dicas do comportamento por meio do método *IT()* de uma forma descritiva, o *CHAI* é usado com seus métodos de *EXPECT()* para fazer asserção acerca da resposta esperada, em que o servidor envia de volta na requisição através do método HTTP *POST*.

No teste isso pode ser observado na função *endHandler()*, chamada ao final da execução do *superagent* que é um módulo usado para fazer requisições AJAX(*Asynchronous Javascript and XML*, - *Javascript* Assíncrono e *XML*) ao servidor.

Uma forma para usar o *superagente* é: primeiro escolhe-se o tipo da requisição, em seguida envia-se os dados ou corpo e ao final cria-se uma função de *CALL-*

BACK endHandler(). Nesse caso, ela é quem trata de encapsular as asserções dos teste de unidade. Tendo o *MOCHA* instalado no ambiente de desenvolvimento, basta rodar no terminal o comando *MOCHA* aquivoDeTest.js. como na Figura 2, referente ao código fonte do projeto, encontrado em sua forma completa no endereço <https://github.com/Pompeu/rede/tests/area.test.js>.



```
pompeu@pompeu-System-Product-Name:~/Documents/dev/node/rede/tests$ mocha area.test.js

testing Area api restful testing
  ✓ expect Area create on db (77ms)
  ✓ expect post without body response stats false
  ✓ expect get one Area by id from db
  ✓ expect get all Area from db (39ms)
  ✓ expect update one Area with id (42ms)
  ✓ expect delete one Area by id form db

6 passing (253ms)

pompeu@pompeu-System-Product-Name:~/Documents/dev/node/rede/tests$
```

Figura 2. Resultado dos Testes

3. Testes Assíncronos

O *NODEJS* em sua essência arquitetural, possui comportamento assíncrono e isso gera influência direta nos testes de comportamentos criados, pelo simples fato de trabalhar com conceitos de *Event loop*. Trata-se de um *loop* infinito que a cada interação faz checagem na fila de evento e os resolve. Conforme descrito em Pereira.

”Um servidor *NODEJS* utiliza o mecanismo *Event loop*, sendo responsável por lidar com a emissão de eventos. Na prática, a função `http.createServer()` é responsável por levantar um servidor e o seu *callback function(request, response)* apenas é executado quando o servidor recebe uma requisição.”(Pereira 2014)

A suíte de testes que *NODEJS* proporciona de forma nativa, bem simples e documentada que pode ser verificada e estudada no link da própria documentação <http://nodejs.org/api/assert.html>, contudo ela possui limitações nas quais inviabiliza e causa complexidade ao desenvolvimento guiado por comportamentos. Esse foi o fator da escolha do *CHAI* para tal trabalho de descrever e deixar cada asserção feita mais compressível e próxima do comportamento em questão, pois em geral uma suíte de testes autonomizados para ambiente *REST* se utiliza de dados fictícios(*Mocks*) no envio de requisições e em suas respostas, para com isso gerar testes de forma mais simples. Contudo, tal prática pode deixar com que comportamentos reais passem despercebidos como lentidões. A exemplo, da aplicação Rede de Conhecimentos, todos os *endpoints*(rotas *REST*), são testadas com o *NODEJS* e o banco de dados *NEO4J*(*NEO4J* é uma categoria de bases de dados *NoSQL* de grafos Bruggen (2014)) em plena execução no ambiente de desenvolvimento, isso demonstra que os testes são feitos diretamente usando requisições *HTTP* e armazenamento no banco de dados real dando confiabilidade ao sistema.

Essa abordagem que a suíte de teste proporciona pelo *NODEJS* cria independência quanto ao uso da tecnologia para criação da aplicação *RESTful*. A mesma metodologia de teste pode ser usada para testar o comportamento independente da tecnologia, pois uma

requisição *HTTP* mesmo implementada em outras tecnologias como o *JAVA*, *PYTHON* ou outros, segue a especificação do protocolo *HTTP*, e com isso possuem implementações semelhantes, sendo passíveis de testes.

Os teste assíncronos, são bons para testar rotas de aplicações do tipo *REST*, pois os pacotes que trafegam sobre o protocolo *HTTP* podem ser pedidos e terem atrasos na resposta. O *MOCHA* tem a capacidade de testar comportamentos assíncronos, basta usar o argumento *done* na função de retorno (*callback*) no metodo *it()*, no código presente no quadro 02 pode ser visto o teste.

Quadro 2. Código de teste para api rest assincronas

```
1
2 it('Deve Criar um aluno no sistema',function (done) {
3     function endHandler (err , res) {
4         err.should.not.exist;
5         res.should.be.ok;
6         res.body.should.have.property("name");
7         res.body.should.have.property("email");
8         res.body.should.have.property("cpf");
9         done();
10    }
11    superagent
12        .post(url.resolve(baseUrl , 'alunos '))
13        .send(body)
14        .end(endHandler);
15
16    });
```

Esse caso de teste é usado para fazer asserção no método *should*, deixando ainda mais simples o entendimento do comportamento esperado, o método *done()* é chamado no fim da execução de todos os testes unitários dentro da função *endHandler()*. É ele quem controla o comportamento assíncrono da requisição do *superagent* de forma que quando estiver finalizado ele emite sinais ao *Event loop* indicando o final de todos os testes unitários.

3.1. Testes de Serviços RESTful

”Testes de serviços web *RESTful* devem levar em consideração três aspectos particularmente importantes presentes nessa tecnologia: i: insuficiência semântica do documento de descrição do serviço; ii) variedade de formatos para representação dos recursos; iii) uso de requisição *HTTP* para execução dos serviços [Canfora e Penta 2009 apud Souza e Correa , 2012]”

Sempre levando em consideração esses três aspectos, o desenvolvimento guiado pelos testes de comportamento para serviços *RESTful* é capaz de diminuir a insuficiência semântica em cada caso teste. Isso se deve ao fator primordial do *BDD* ser capaz de criar proximidade entre desenvolvedores e os demais interessados com a excelência do sistema, demonstrando que principal objetivo dele é criar isso proximidade. Contudo o uso

de requisições do tipo *HTTP* possuem a necessidade de que o serviço esteja em execução fazendo com que os testes sejam limitados a ambientes de desenvolvimentos, pois técnicas para testar sistemas em produção são outras, fora do escopo atual.

4. Conclusão

Como o crescimento de metodologias ágeis, em conjunto ao *TDD* e *BDD*, o desenvolvimento de sistemas fica constantemente sofrendo mudanças em busca de melhorias nas técnicas de manutenção, criação e até mesmo na participação comum de todos envolvidos acerca de determinado projeto.

Ficou claro que o desenvolvimento guiado pelos testes a partir dos comportamentos esperados, foi de fato uma boa escolha, pois isso trouxe proximidade entre *stackholder* e desenvolvedor, criando um elo de credibilidade e confiança, para dar continuidade ao longo do desenvolvimento. É visível à todos os participantes do projeto que testar um sistema de forma automatizada com ferramentas como *MOCHA* e *CHAI*, traz grandes ganhos ao produto final.

Com o tempo ficou perceptível que cada serviço *RESUFul* da aplicação estava devidamente documentado e testado. Contudo no início da criação dos testes acerca dos comportamentos esperados, acreditava-se que o fator tempo seria dispendioso e cansativo. Porém isso foi diferente pelo simples motivo de quando um comportamento é testado, esse teste tem um poder grandioso quando observado de forma global, porque é testado desde a entrada de dados através do método *POST*, até a atualização dos dados *PUT*, remoção *DELETE* e recuperação de dados *GET*, que são os verbos *HTTP* indicados para aplicações do tipo *CRUD*.

Referências

- Saudate, Alexandre (2013). *REST*, Construa API's inteligentes de maneira simples.
- Aniche, Mauricio (2014). *Test-Driven-Development*, Teste e Design no Mundo Real.
- Hugo, Barauna (2013). *Cucumber e Rspec*, Construa aplicacoes Ruby On Rails com testes e especificacoes.
- Dan, North (2009). *Introducing bdd*, <http://dannorth.net/introducing-bdd/> Acesso em: 01 junho 2015.
- NodeJS.org (2015). *Node JS*, <https://nodejs.org/> Acesso em: 02 junho 2015.
- Mocha.org (2015) *Mocha JS*, <http://mochajs.org/> Acesso em: 02 junho 2015.
- Chai.com (2015). *Chai JS*, <http://chaijs.com/> Acesso em: 02 junho 2015.
- Ribeiro, Pereira Caio (2014). *Node.js*, Aplicacoes web *real-time* com Node.js.
- Rik Van Bruggen (2014) *Learning Neo4j*, Run blazingly fast queries on complex graph datasets with the power of the Neo4j graph database.
- Thiago Silva de Souza, Alexandre Luis Correa, Antonio Juarez Alencar, Eber Assis Schmitz (2012). *Uma Abordagem Baseada em Especificacoes para Testes de Web Services RESTful*.