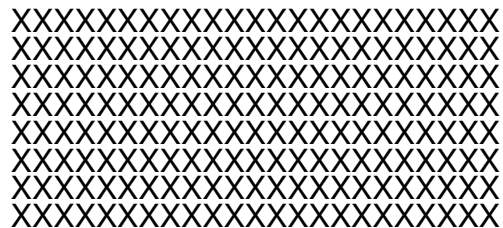


[illegible]

*GraphDB, noSQL , NODEJS , JavaScript*



O termo BDD foi cunhado por Dan North em meados de 2003. Nessa época Dan North teve demasiada dificuldade em ensinar TDD a seus alunos, e por esse motivo ele evoluiu e simplificou o desenvolvimento guiado por testes. 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\*.

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 1 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.

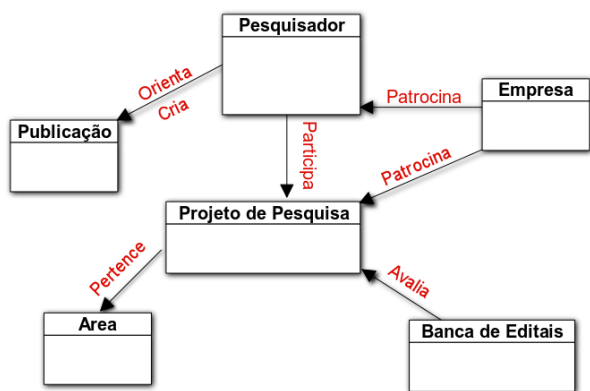


Figure 1: Diagrama de Conceitual da Rede de Conhecimentos

### 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.

"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, Maurício, p.28)

### 2.3 Metodologias

Apesar de serem utilizados métodos ágeis para o desenvolvimento do sistema em comunhão ao *BDD*, para a pesquisa acerca dos conceitos e tecnologias usadas nesse projeto foram usados livros livres em várias oportunidades e disponibilizados nos sites das tecnologias usadas, como a documentação do *NODEJS* disposta em sua página, assim como do *MOCHA* e *CHAI*, 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.

Ele possui vasta documentação e exemplos em seu site oficial (<http://chaijs.com/>), 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.

#### Listing 1: Código para Testar uma Rota do Tipo Post no Sistema

```
var expect = require('chai').expect,
    superagent = require('superagent'),
    ch = require('charlatan'),
    url = require('url'),
    baseUrl = 'http://localhost:3000/api/area';

describe('testing area api restful',
function () {
    var id = null;
    var body = {
        nomeArea : ch.Name.name(),
        subCodigo : ch.rand(8000, min = 7001),
        subArea : ch.Name.name(),
        subNivel : ch.Name.title(),
    }
    it('expect area create on db',
    function (done) {
        function endHandler(err, res) {
            expect(err).to.not.exist;
            expect(res).to.exist;
            expect(res.body.status).to.true;
            expect(res.body.err).to.null;
            expect(res.body.result)
                .to.an('object');
            expect(res.body.result.id)
                .to.an('Number');
            id = res.body.result.id;
            expect(id).to.an('Number');
            done();
        }
        superagent
            .post(url.resolve(baseUrl, 'area'))
            .send(body)
            .end(endHandler);
    })
})
```

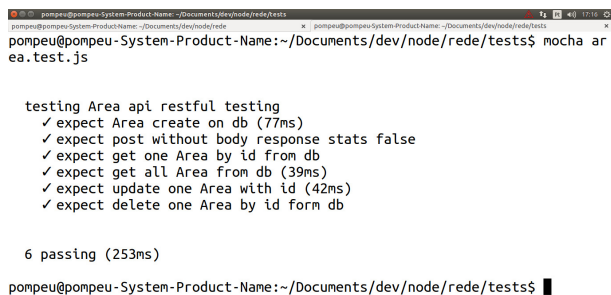
```
});
});
```

Esse código exemplifica e demonstra a clareza na qual o teste é escrito, de forma simples e sucinta.

Ele 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 *superagent* é: primeiro escolhe-se o tipo da requisição, em seguida envia-se os dados ou corpo e ao final cria-se uma função de *CALLBACK endHandler()*. Nesse caso, ela é quem trata de encapsular as asserções dos testes de unidade. Tendo o *MOCHA* instalado no ambiente de desenvolvimento, basta rodar no terminal o comando *MOCHA* *acervoDeTest.js*. como na figura 2, referente ao código fonte do projeto, encontrado em sua forma completa no endereço <https://github.com/Pompeu/rede/blob/master/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 from db

6 passing (253ms)
pompeu@pompeu-System-Product-Name: ~/Documents/dev/node/rede/tests$
```

Figure 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 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.

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, Caio Ribeiro 2014, p.14)

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 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. E como exemplo, na aplicação da Rede de Conhecimento, todos os *endPoints* (rotas *REST*), são testadas com o servidor *NODEJS* e o banco de dados *NEO4J* em plena execução e isso demonstra que os testes são feitos diretamente usando requisições *HTTP* e armazenamento no banco de dados real. 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 testes 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 método *it()*, no código exemplo número 2 pode ser visto o teste.

```
it('Deve Criar um aluno no sistema',
  function (done) {

function endHandler (err, res) {
  err.should.not.exist;
  res.should.be.ok;
  res.body.should.have.property("name");
  res.body.should.have.property("email");
  res.body.should.have.property("cpf");
  done();
}

superagent
  .post(url.resolve(baseUrl, 'alunos'))
  .send(body)
  .end(endHandler);

});
```

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 im-

portantes 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.

Contudo o uso de requisições HTTP possui a necessidade de que o serviço esteja em execução fazendo com que os testes sejam limitados a ambientes de desenvolvimentos em alguns casos, 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.

No início desse do projeto descrito nesse artigo foram pensadas várias metodologias para agilizar, documentar e melhorar a consistência do mesmo. 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.

## 5. REFERENCES

- [1] Saudate, Alexandre, REST, Construa API's inteligentes de maneira simples, 2013.
- [2] Aniche Maurício, Test-Driven-Development, Teste e Design no Mundo Real, Aniche Maurício, 2014.
- [3] Ribeiro , Pereira Caio, Node.js, Aplicações web real-time com Node.js , Ribeiro , Pereira Caio, 2013.
- [4] Baraúna , Hugo, Cucumber e RSpec, Construa aplicações Ruby e Rails com testes e especificações ,Baraúna , Hugo, 2014.
- [5] Aniche Maurício, Test-Driven-Development .NET, Teste e Design no Mundo Real .NET , Aniche Maurício, 2014
- [6] Cody Lindley, JavaScript Enlightenment, First Edition, based on JavaScript 1.5, ECMA-262, Edition 3, 2003

- [7] Addy Osmani, JavaScript Design Patterns, Learning JavaScript Design Patterns, 2014
- [8] Colin J. Ihrig, Pro Node.js for Developers, Pro Node.js for Developers, 2013
- [9] Rik Van Bruggen, Learning Neo4j, Run blazingly fast queries on complex graph datasets with the power of the Neo4j graph database, 2014
- [10] Rik Van Bruggen, Learning Neo4j, Run blazingly fast queries on complex graph datasets with the power of the Neo4j graph database, 2014
- [11] Thiago Silva de Souza, Alexandre Luis Correa, Antonio Juarez Alencar, Eber Assis Schmitz, Uma Abordagem Baseada em Especificações para Testes de Web Services RESTful.