

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE
SISTEMAS

LUIZ CEZER MARRONE FILHO

**TDD ON RAILS - DESENVOLVIMENTO GUIADO A TESTES EM
APLICAÇÕES WEB COM FRAMEWORK RAILS**

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA

2012

LUIZ CEZER MARRONE FILHO

**TDD ON RAILS - DESENVOLVIMENTO GUIADO A TESTES EM
APLICAÇÕES WEB COM FRAMEWORK RAILS**

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas – COADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Juliano Rodrigo Lamb

MEDIANEIRA

2012



TERMO DE APROVAÇÃO

TDD ON RAILS - DESENVOLVIMENTO GUIADO A TESTES EM APLICAÇÕES WEB COM FRAMEWORK RAILS

Por

Luiz Cezer Marrone Filho

Este Trabalho de Diplomação (TD) foi apresentado às **09:10h** do **dia 03 de Julho de 2012** como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, Câmpus Medianeira. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. Juliano Rodrigo Lamb
UTFPR – Câmpus Medianeira

Prof. Fernando Schütz
UTFPR – Câmpus Medianeira

Prof. Diego Stiehl
UTFPR – Câmpus Medianeira

Prof. Juliano Rodrigo Lamb
UTFPR – Câmpus Medianeira

RESUMO

FILHO, M. Luiz Cezer. TDD on Rails - Desenvolvimento guiado a testes em aplicações web com framework Rails. 2012. Trabalho de conclusão de curso (Tecnologia em Análise e Desenvolvimento de Sistemas), Universidade Tecnológica Federal do Paraná. Medianeira 2012.

O uso de técnicas de teste no desenvolvimento de aplicações proporciona maior segurança e confiabilidade no código que é gerado, pois o mesmo é testado a cada nova funcionalidade que é criada. Esse trabalho tem como objetivo utilizar o TDD como técnica de teste em aplicações web utilizando *framework Rails* e, ao final desta, desenvolver uma aplicação utilizando as técnicas de TDD e os testes do próprio *framework Rails* para testar a confiabilidade do código gerado e a funcionalidade da aplicação, como estudo experimental.

Palavras-chave: *Ruby*. Desenvolvimento Web. TDD

ABSTRACT

FILHO, M. Luiz Cezer. TDD on Rails - Test Driven Developement with framework Rails for web applications. 2012. Completion of course work (Technology Analysis and Systems Development), Federal Technological University of Paraná,. Medianeira 2012.

The use of testing techniques in the development of applications provides greater security and reliability in the code that is generated, as it is tested every new feature that is created. This paper aims to use the TDD as a technique for testing web applications using the Rails framework and at the end of this, developing an application using the techniques of TDD and testing Rails framework itself to test the reliability of the generated code and application functionality as experimental study.

Palavras-chave: *Ruby. Web Development.TDD*

LISTA DE FIGURAS

Figura 1 - Qualidade de Software.....	16
Figura 2 - Exemplo de código ruby.....	19
Figura 3 - Como funciona o MCV do Rails	21
Figura 4 - Estrutura de uma aplicação Rails	22
Figura 5 - Criação de Aplicação Rails e Diretórios para Testes	24
Figura 6 - Arquivo database.yml.....	25
Figura 7 - Criação de um Model e arquivos de fixtures na aplicação	26
Figura 8 - Fixture com dados para testes	26
Figura 9 - Exemplo de classe de teste unitário.....	27
Figura 10 - Exibindo execução do teste	28
Figura 11 - Exemplo de Teste Funcional.....	29
Figura 12 - Executando os testes funcionais.....	30
Figura 13 - Criando um test integration	30
Figura 14 - Exemplo de teste de integração.....	31
Figura 15 - Executando teste de integração.....	31
Figura 16 - Caso de Uso aplicação	33
Figura 17 - Criando Aplicação	34
Figura 18 - Criação de Model User	34
Figura 19 - Criando Scaffold Task.....	35
Figura 20 - Teste de validação de dados model User	36
Figura 21 - Teste falhando	37
Figura 22 - Mínimo de código para o teste passar	37
Figura 23 - Testes sendo executados sem erros	38
Figura 24 - Teste de validação de número de caracteres	38
Figura 25 - Teste de número de caracteres não dispara erro	39
Figura 26 - Ajuste no model	39
Figura 27 - Teste de número de caracteres falhando.....	40
Figura 28 - Ajustando teste de número de caracteres.....	40
Figura 29 - Todos os testes passam para o model User	41
Figura 30 - Esqueleto de testes funcionais	42
Figura 31 - Utilização de filtro no controller	42
Figura 32 - Método autorize no controller principal	43
Figura 33 - Ações responsáveis por login e logout de usuário	43
Figura 34 - Método authenticate.....	44
Figura 35 - Erros no teste funcional	44
Figura 36 - Método que simula autenticação de usuário	45
Figura 37 - Inserindo método login_as aos testes.....	45
Figura 38 - Teste funcionais funcionando	46
Figura 39 - Criando teste de integração	47

Figura 40 - Testes de integração para login e logout	47
Figura 41 - Teste de fluxo normal da aplicação.....	48
Figura 42 - Resultados para os testes de integração	49
Figura 43 - Inserindo usuário via console.....	50
Figura 44 - Acesso a aplicação sem autenticação	51
Figura 45 - Aplicação com usuário autenticado.....	51
Figura 46 – Ajuste de view de listagem de tarefas	52
Figura 47 - Ajuste no template principal da aplicação	52
Figura 48 - Tela de login	53
Figura 49 - Tela de adição de nova Task	53

LISTA DE SIGLAS

COC	<i>Convention over Convencion</i>
DRY	<i>Don't Repeat Yourself</i>
MVC	<i>Model-View-Control</i>
ROR	<i>Ruby on Rails</i>

SUMÁRIO

1 INTRODUÇÃO	7
1.1 OBJETIVO GERAL	7
1.2 OBJETIVOS ESPECÍFICOS	8
1.3 JUSTIFICATIVA	8
1.4 ESTRUTURA DO TRABALHO	9
2 REVISÃO BIBLIOGRÁFICA	11
2.1 DESENVOLVIMENTO NÃO GUIADO POR TESTES	11
2.2 METODOLOGIAS ÁGEIS	12
2.3 METODOLOGIA XP	14
2.4 QUALIDADE DE SOFTWARE	15
2.5 TDD	16
2.6 LINGUAGEM RUBY	18
2.7 FRAMEWORK RAILS	20
2.8 TESTE COM FRAMEWORK RAILS	23
2.8.1 OS TRÊS AMBIENTES	25
2.8.2 FIXTURES	25
2.8.3 ASSERTS	27
2.8.4 TESTES UNITÁRIOS	27
2.8.5 TESTES FUNCIONAIS	28
2.8.6 TESTE DE INTEGRAÇÃO	30
3 MATERIAIS E MÉTODOS	32
3.1 FERRAMENTAS UTILIZADAS	32
3.2 ESTUDO EXPERIMENTAL	33
3.2.1 Criando aplicação e principais recursos	34
3.2.2 Criando testes unitários para model User	35
3.2.3 Criação de teste funcional para <i>controller</i> Task	41
3.2.4 Teste de integração	46
4 RESULTADOS E DISCUSSÕES	50
5 CONSIDERAÇÕES FINAIS	55
5.1 CONCLUSÃO	55

5.2 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO	55
6 REFERÊNCIAS BIBLIOGRÁFICAS	56

1 INTRODUÇÃO

O desenvolvimento de software não é uma tarefa fácil, uma vez que muitos fatores devem ser considerados e avaliados antes mesmo de se iniciar o projeto de um software, como viabilidade do projeto, prazo de entrega, linguagens e frameworks a serem utilizados e metodologia a ser utilizada.

Os problemas geralmente surgem durante o decorrer desse desenvolvimento e podem acabar causando um retrabalho no projeto para a correção desses problemas. Dentre os principais problemas está a correção de bugs que ocorrem à medida que o software cresce e as linhas de código aumentam, nessas horas entram testes e mais testes para verificação e correção dos problemas ocorridos, o que prejudica e muito o andamento do projeto.

O TDD (Test Driven Development) ou Desenvolvimento Guiado a Testes visa minimizar esses problemas e a perda de tempo de debug de um projeto, adotando uma filosofia simples “teste antes, codifique depois”. Adotando essa metodologia o programador escreve seus testes antes, após isso faz o seu código mais simples para passar no teste, depois o código é refatorado e novamente testado, ao final dos testes o desenvolvedor já tem um código final pronto, testado e escrito da melhor maneira para atender o determinado requisito (SANCHEZ, 2006).

Outros pontos importantes a serem citados sobre o TDD é que ele prega a idéia de “passos de bebê” onde o software vai crescendo em pequenas partes, em pequenos passos até ter sua construção completada (SANCHEZ, 2006).

Esse trabalho tem como objetivo realizar o desenvolvimento de uma aplicação utilizando TDD como metodologia de testes a fim de apresentar seus benefícios, vantagens e desvantagens.

1.1 OBJETIVO GERAL

Apresentar as vantagens do TDD por meio de um estudo experimental, aplicando a técnica no desenvolvimento de uma aplicação web com *framework* Rails.

1.2 OBJETIVOS ESPECÍFICOS

- a) Descrever como é o desenvolvimento onde os testes são realizados após a codificação e quais problemas podem surgir com essa prática.
- b) Conceituar, por meio de referencial teórico, metodologias ágeis de desenvolvimento e qualidade de software.
- c) Desenvolver referencial teórico sobre TDD, seus benefícios e como desenvolver *software* adotando essa filosofia.
- d) Criar uma aplicação para mostrar de maneira prática como funciona o TDD no desenvolvimento de aplicações, realizando testes antes do código final.

1.3 JUSTIFICATIVA

Os riscos de se desenvolver aplicações sem fazer os testes necessários e da melhor maneira podem ser muito grandes, os problemas vão desde pequenos *bugs* no projeto a grandes problemas que podem inviabilizar o uso da aplicação.

Em algumas situações o *software* é colocado em produção, mas seu código está tão mal estruturado ou mal testado que inviabiliza seu uso. Outro ponto é que com o passar o tempo à manutenção do se torna complexa ou os custos para realizá-la são altos sendo que é mais fácil substituir o *software* do que realizar a manutenção. (RIBEIRO, 2010)

O tempo gasto realizando *debug* em uma aplicação para encontrar uma falha pode ser muito grande e custar um tempo precioso para a entrega de um projeto no seu determinado prazo, além disso, pequenos erros que não foram testados podem se propagar gerando erros em outros pontos da aplicação.

O desenvolvimento guiado por testes visa melhorar o código e minimizar esses problemas, pois os testes são todos feitos antes da codificação final e o código é sempre refatorado ficando mais limpo, funcional e de fácil manutenção futura, ao final dos testes o programador já tem um código final testado e funcionando corretamente na aplicação, o que pode poupar muito tempo em correção de *bugs*. Segundo (BECK, 2002) citado por (RIBEIRO, 2010):

TDD mantém o programador focado na solução, de forma que o software não fica carregado de códigos desnecessários, duplicados ou de difícil manutenção, impedindo a deterioração do sistema.

O uso da técnica de TDD também traz benefícios ao desenvolvedor, pois encoraja o mesmo a sempre melhorar seu código por meio dos testes, fazendo o mesmo adquirir mais experiência como desenvolvedor, melhorando seu código e sua maneira de pensar, aumentando assim sua produtividade na hora de escrever as aplicações. Segundo (BECK, 2002):

Os testes lhe dão a confiança de que grandes refatorações não mudarão o comportamento do sistema, o que se conclui que, quanto maior a confiança, mais agressivamente você poderá conduzir refatorações em larga escala que estenderão a vida de seu sistema. A refatoração torna a elaboração dos próximos testes muito mais fácil.

Escrever testes para software é uma maneira entender melhor os requisitos, minimizar os riscos de falha, aumentar a confiança dos programadores e ainda assegurar uma melhor qualidade do *software* desenvolvido. A utilização de TDD como metodologia de testes pode ser facilmente posta em prática no desenvolvimento web com o *framework* Rails que além de agilizar o desenvolvimento de aplicações também favorece a criação e execução de testes dentro da aplicação.

1.4 ESTRUTURA DO TRABALHO

O trabalho se divide em seis capítulos, sendo que o primeiro trata da contextualização do tema abordado, definindo os objetivos do trabalho e justificativas pela escolha do tema.

O segundo capítulo trata do embasamento teórico, onde são tratados assuntos relacionados e as tecnologias a serem utilizadas no estudo experimental. O terceiro apresenta um estudo experimental onde é desenvolvida uma aplicação com utilização de técnicas de testes e ao final do desenvolvimento aponta-se as vantagens e conclusões sobre a aplicação desenvolvida. O quarto capítulo apresenta a discussão sobre os resultados obtidos.

O quinto capítulo apresenta as considerações finais do trabalho. E por fim, no sexto capítulo é feita a referência bibliográfica sobre os assuntos abordados no trabalho.

2 REVISÃO BIBLIOGRÁFICA

Esse capítulo tem como objetivo apresentar os conceitos sobre as tecnologias e metodologias a serem utilizadas no trabalho.

2.1 DESENVOLVIMENTO NÃO GUIADO POR TESTES

Desenvolvimento não guiado a testes pode ser dito como a prática de desenvolvimento onde o desenvolvedor não aplica nenhum tipo de testes ao seu software enquanto desenvolve ou também quando os testes criados não são bem elaborados e não testam o comportamento adequando das funcionalidades do software.

Desenvolver *software* é uma tarefa complicada, são muitas questões que devem ser observadas antes de iniciar seu desenvolvimento. É necessário estudar sobre o problema, escolher a melhor opção em tecnologia, estudar a viabilidade do projeto, metodologia a ser seguida.

Além desses fatores outro fator que acaba sendo deixado de lado quando o projeto está em fase de codificação é o uso das boas práticas em programação, não existe uma regra geral, mas é interessante o programador utilizar algumas regras na hora de codificar não só para que o mesmo possa entender seu código, mas para que outros possam dar manutenção no mesmo futuramente.

Um dos pontos mais importantes dentro das boas práticas está nos teste sobre as funcionalidades da aplicação.

Muitos programadores possuem o hábito de codificar e testar, caso o código funcione ele passa para outra etapa do desenvolvimento. O grande problema é que à medida que a aplicação cresce esse trecho de código até então funcional pode causar problemas em outras partes da aplicação como um efeito dominó, o que trará muitos problemas não só ao desenvolvedor em questão, mas ao projeto todo, além do fato que esse código mal escrito pode possuir duplicações ou linhas que não fazem diferença nenhuma na eficiência ou resultado final do código, usando da regra

“funcionou está bom” o programador não refatora seu código, deixando o mesmo poluído e com trechos inúteis.

Existem programadores que fazem testes, mas o problema é a maneira como esses testes são feitos, na maioria das vezes não são bem feitos e não testam o comportamento adequando da funcionalidade.

O desenvolvimento não guiado a testes pode a princípio pode parecer o mais ágil ou o até mesmo o mais correto, pois não “se perde tempo” escrevendo testes, mas problemas podem surgir depois e quando surgem podem afetar qualquer ponto da aplicação em questão.

É com base nessas situações e problemas que o desenvolvimento guiado a testes trabalha, não é necessário que o projeto esteja 100% testado, mas é necessário ter testado e garantido a funcionalidade pelo menos os principais pontos da aplicação.

2.2 METODOLOGIAS ÁGEIS

Para amenizar os problemas do desenvolvimento de *software* sem planejamento é necessário adotar uma metodologia que ajude no processo de planejamento e estudo do problema a ser solucionado.

Segundo (FOWLER, 2003) citado por (PASSUELLO, 2005) “Metodologias impõem um processo disciplinado no desenvolvimento de software, com o objetivo de torná-lo mais previsível e mais eficiente”.

Mas o problema é que muitas dessas metodologias são muito burocráticas exigindo uma pesada documentação e gerenciamento do projeto, podendo acabar sendo vistas como atrasos no projeto, já que exigem um grande tempo sendo gasto em análise do *software* a ser construído como na documentação de cada uma das ações do mesmo. (FOWLER, 2003)

Surgiram então nos anos noventa uma forma de reação a essas metodologias são as chamadas metodologias ágeis no processo de desenvolvimento de *software*. (BECK, 2001)

Criado por Kent Beck juntamente com outros quinze desenvolvedores de *software* com grande experiência, o Manifesto Ágil é o documento que reúne os

princípios e praticas dessa metodologia dentro do processo de desenvolvimento. (BECK, 2001). O Manifesto Ágil possui alguns valores que são levados em conta:

- **Indivíduos e interações** mais que processos e ferramentas
- **Software em funcionamento** mais que documentação abrangente
- **Colaboração com o cliente** mais que negociação de contratos
- **Responder a mudanças** mais que seguir um plano
- Além dos valores o Manifesto Ágil se baseia em princípios (BECK, 2001):
- Satisfazer o cliente por meio de entregas adiantadas e contínuas do *software*
- Aceitar mudanças de requisito a qualquer momento dentro do projeto
- Entrega de software em funcionamento com frequência semanal ou mensal
- Pessoas que entendem do negócio e desenvolvedores devem estar sempre trabalhando juntas durante o processo de desenvolvimento
- Trabalhar com pessoas motivadas no projeto, dando a eles um bom ambiente de trabalho e confiar que farão aquele que foi designado a eles
- A melhor maneira de se transmitir informação é por meio de uma conversa pessoal
- Software funcional é a medida primária de progresso
- Processos ágeis promovem um ambiente sustentável
- Contínua atenção a excelência técnica e bom design
- Simplicidade
- As melhores arquiteturas, requisitos e designs emergem de times auto-organizáveis
- Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e aperfeiçoam seu comportamento de acordo.

Por meio desses princípios e valores pode-se notar que as metodologias ágeis deixam de lado a documentação pesada e processos de extração e elaboração de requisitos e passam a valorizar mais o *software* funcionando, liberando pequenas versões à medida que as mesmas ficam prontas, trazem o cliente para dentro do processo de desenvolvimento fazendo o mesmo participar e

acompanhar o crescimento do projeto enfatiza interação entre pessoas sejam elas do time de desenvolvimento ou pessoas que entendem do negócio e abraçam mudanças no *software* a qualquer momento. (FOWLER, 2003)

2.3 METODOLOGIA XP

A Extreme Programming mais conhecida com XP é uma metodologia de desenvolvimento que tem como finalidade entregar *software* de qualidade e com as necessidades que o cliente realmente precisa.

É uma metodologia voltada para pequenos e médios times de desenvolvimento que trabalham em projetos orientados a objetos e que seus requisitos podem sofrer constantes mudanças (KUHN, 2009).

A satisfação do cliente é o que faz a metodologia ser bem sucedida. O XP busca o máximo desempenho da equipe mantendo um ritmo saudável de trabalho onde horas extra só são realizadas se forem agregar algo de valor ao produto final. O cliente obterá seu produto em pequenos lançamentos, podendo utilizá-lo, aprender com o mesmo e avaliar se o que foi desenvolvido era o desejado. (TELES, 2008). O XP se baseia em valores que são (TELES, 2008):

- Feedback
- Comunicação
- Simplicidade
- Coragem
- Respeito

São esses valores que ditarão com a equipe deve se comportar durante o desenvolvimento do projeto (TELES, 2008). Juntamente com os valores existem também princípios que servem para complementar os valores (TELES, 2008):

- Feedback rápido
- Presumir simplicidade
- Mudanças incrementais
- Abraçar mudanças
- Trabalho de alta qualidade

Aliado aos valores e aos princípios a metodologia XP também possui algumas práticas para descrevem quais atitudes a equipe deve tomar no ambiente de trabalho e como se deve ocorrer o processo de desenvolvimento pela equipe.

- *Design* Simples
- Programação em Pares
- Código de posse coletiva
- Desenvolvimento Guiado a testes
- Ritmo Sustentável

2.4 QUALIDADE DE SOFTWARE

Segundo (PRESSMAN, 1995) “Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo *software* profissionalmente desenvolvido”.

A qualidade de *software* não pode estar ligada somente ao *software* fazer algo certo, como foi pedido pelo cliente e também não se restringe ao *software* final criado. Qualidade de *software* está ligada também a qualidade do código, reusabilidade, manuteabilidade, aos processos adotados durante o desenvolvimento como análise de requisitos e documentação. (PRESSMAN, 1995)

Existem fatores para determinar a qualidade de *software* (PRESSMAN, 1995):

- **Corretitude:** O *software* satisfaz as especificações determinadas e cumpre os objetivos desejados pelo cliente.
- **Confiabilidade:** O *software* executar determinada funcionalidade com a precisão esperada.
- **Eficiência:** Quantidade de recursos computacionais e de código exigida pelo *software* para executar uma funcionalidade.
- **Integridade:** Controlar acesso ao *software* ou a dados a pessoas não autorizadas.

- **Usabilidade:** Esforço para aprender, operar e preparar a entrada e interpretar uma saída do *software*.
- **Manuteabilidade:** Esforço para localizar e reparar erros no *software*.
- **Flexibilidade:** Esforço para modificar um programa operacional.
- **Testabilidade:** Esforço exigido para testar um *software* a fim de garantir que ele execute uma funcionalidade pretendida.
- **Portabilidade:** Esforço exigido para transferir o programa de um ambiente de sistema de *hardware/software* para outro.
- **Reusabilidade:** À medida que um *software* cresce possibilita algumas de suas partes serem reutilizadas em outros ou até no mesmo *software*.
- **Interoperabilidade:** Esforço exigido para se acoplar um sistema a outro.

Essas são os fatores propostos por McCall, Richards e Walters (1977) e citados por (PRESSMAN, 1995). Esses fatores podem ser agrupados e representados conforme a Figura 1.



Figura 1 - Qualidade de Software

Fonte: (TestExpert, 2008)

2.5 TDD

TDD é a prática de desenvolvimento de *software* onde os testes são criados antes do código final, com o intuito de que ao final de todos os testes necessário o

programador tenha uma aplicação funcional, com uma estrutura bem feita e com todos os testes necessários realizados. (BASTOS, 2012)

Os principais motivos que levam a utilização do TDD são que ele traz benefícios tanto o *software* produzido quanto ao programador que desenvolve segundo essa prática (MAFRA, 2009).

A principal característica está na forma como ele influenciará o programador a desenvolver seu código, que seria fazendo os testes antes, codificando depois e refatorando sempre que possível.

A prática de TDD pode ser resumida de maneira prática em três passos (MAFRA, 2009):

- **Escrever um teste que falhe** – primeiro deve-se pensar no que fazer e logo após isso definir como executar e o resultado que se espera.
- **Fazer o teste passar** – escrever o suficiente para o código passar
- **Refatorar** – se o que foi escrito para o teste passar puder ser melhorado, ele deve ser melhorado, sem que altere o comportamento do teste.

Logo de início realizar testes pode parecer inútil para o programador e é esse o ponto crítico onde o programador precisa se reeducar para adotar TDD no seu trabalho e usufruir dos benefícios do desenvolvimento guiado a testes (MAFRA, 2009). Segundo (BECK, 2002):

O TDD faz com que o programador ganhe confiança sobre seu código com o passar do tempo, isto é, à medida que os testes vão se acumulando, ele ganha confiança no comportamento do sistema. E ainda, à medida que o desenho é refinado, mais e mais mudanças se tornam possíveis.

Dentro do TDD ainda existem alguns conceitos e práticas e uma delas é a prática em forma de “*baby steps*” que pode ser definida com um crescimento pouco a pouco do projeto. (ANICHE, 2010)

Segundo (SANCHEZ, 2006) “*Baby steps* (passos de bebê) é um termo que expressa como o desenvolvimento incremental proposto no TDD deve ser feito. A ideia é implementar pouco a pouco, para que a cada pequena mudança no *software* seja possível obter o feedback sobre aquilo que foi implementado.”

Utilizando desse artifício o *software* cresce aos poucos, função por função e como cada uma delas já é testada é possível obter de maneira rápida uma resposta

do sistema se aquilo que está sendo testado é realmente funcional e está sendo feito da melhor maneira. (SEA, 2009)

Mesmo com os benefícios citados é difícil avaliar se a prática de TDD no desenvolvimento é realmente tão eficiente a ponto de mudar a qualidade do *software*, ou diminuir a correção de erros. (ANICHE, 2010)

Existem algumas pesquisas que indicam que o TDD ajuda tanto na melhoria de código escrito quanto na redução de problemas do *software* (ANICHE, 2010).

Segundo (JANZEN, 2005) citado por (ANICHE, 2010):

Estudos mostraram que programadores usando TDD na indústria produziram código que passaram em aproximadamente 50% mais testes caixa-preta do que o código produzido por grupos de controle que não usavam TDD. Além do mais, o grupo que usava TDD gastou menos tempo com *debug*. Janzen também mostrou que a complexidade dos algoritmos era muito menor e a quantidade e cobertura dos testes era maior nos códigos escritos com TDD.

Qualidade e manutabilidade de código também são aspectos que são melhorados com o uso de TDD, segundo (LANGR, 2002) citado por (ANICHE, 2010) “TDD aumenta a qualidade código, provê uma facilidade maior de manutenção e ajuda a produzir 33% mais testes comparados a abordagens tradicionais.”

A redução dos erros e de possíveis problemas também é um dos focos do TDD e segundo (M. e WILLIAMS) citado por (ANICHE, 2010) “Um estudo mostrou uma redução de 40-50% na quantidade de defeitos e um impacto mínimo na produtividade quando programadores usaram TDD.”

A escolha de TDD de início pode parecer um atraso no desenvolvimento do *software*, mas ao médio e longo prazo os seus benefícios podem ser visualizados, pois o uso de técnica minimiza riscos, melhorar o código e qualidade do *software*, facilita a documentação, facilita entendimento dos requisitos, torna o sistema mais escalável e de fácil manutenção futura. (BASTOS, 2012) (MAFRA, 2009) (RIBEIRO, 2010)

2.6 LINGUAGEM RUBY

A linguagem foi criada no ano de 1995 no Japão por Yukihiro “Matz” Matsumoto, é uma linguagem orientada a objeto interpretada, de tipagem forte e

dinâmica. Baseada no que havia de melhor de outras linguagens como: Smalltalk, Lisp e Ada, praticamente tudo em *Ruby* é um objeto. (THOMAS, 2008)

É *open-source* o que possibilita aos desenvolvedores abrirem e melhorem o código fonte sempre que julgarem necessário, o que possibilita uma melhoria constante na correção de erros e facilitando a melhoria e inclusão de novas funcionalidades a linguagem. (THOMAS, 2008)

Ruby foi desenvolvida pensando em tentar aproximar ao máximo uma linguagem de máquina de uma linguagem humana esse é um fato que torna a escrita de código *Ruby* mais simples e facilita a leitura até mesmo por pessoas sem conhecimento técnico na área, bastando ter apenas um bom conhecimento em inglês. (Linguagem.)

Tudo dentro da linguagem *Ruby* é um objeto e portanto possui sua classe e seus métodos. Na Figura 2 é possível observar pequenos trechos de código *Ruby* e suas saídas no console.

```
C:\WINDOWS\system32\cmd.exe - irb
C:\Documents and Settings\Cezinha\Desktop>irb
irb(main):001:0> pessoa = "Luiz"
=> "Luiz"
irb(main):002:0> pessoa.class
=> String
irb(main):003:0> pessoa.methods
=> ["%", "select", "[]=", "inspect", "<<", "each_byte", "clone", "gsub", "casecmp", "public_methods", "display", "to_str", "partition", "tr_s", "empty?", "instance_variable_defined?", "tr!", "equal?", "freeze", "rstrip", "match", "grep", "chomp!", "+", "next!", "swapcase", "ljust", "to_i", "swapcase!", "methods", "respond_to?", "upto", "between?", "reject", "sum", "hex", "dup", "insert", "reverse!", "chop", "instance_variables", "delete", "dump", "id", "tr_s!", "concat", "member?", "method", "succ", "find", "eql?", "each_with_index", "strip!", "id", "rjust", "to_f", "singleton_methods", "send", "index", "collect", "oct", "all?", "slice", "taint", "length", "entries", "chomp", "frozen?", "instance_variable_get", "upcase", "sub!", "squeeze", "include?", "send", "instance_of?", "upcase!", "crypt", "delete!", "detect", "to_a", "zip", "lstrip!", "type", "center", "<", "protected_methods", "instance_eval", "object_id", "map", "<=>", "rindex", "any?", "==", ">", "split", "===", "strip", "size", "sort", "instance_variable_set", "gsub!", "count", "succ!", "downcase", "min", "kind_of?", "extend", "squeeze!", "downcase!", "intern", ">=", "next", "find_all", "to_s", "<=" "each_line", "each", "rstrip!", "class", "slice!", "hash", "sub", "tainted?", "private_methods", "replace", "inject", "=~", "tr", "reverse", "nil?", "untaint", "sort_by", "lstrip", "to_sym", "capitalize", "max", "chop!", "is_a?", "capitalize!", "scan", "[]", "unpack"]
irb(main):004:0>
```

Figura 2 - Exemplo de código ruby

Fonte: Autoria Própria

No primeiro comando está sendo criada uma variável com o nome de *pessoa* e lhe atribuindo o valor "Luiz", como tudo em Ruby é um objeto e sua tipagem é dinâmica o uso de aspas duplas no valor de atribuição indica que a variável *pessoa* será do tipo *String*, o comando *pessoa.class* faz essa verificação

e então conclui-se que *pessoa* é uma instancia da classe *String* e possui determinados métodos que são verificados pelo comando `pessoa.methods`.

2.7 FRAMEWORK RAILS

Segundo (MONTEIRO, 2012) “*Ruby on Rails* é um *framework* de desenvolvimento web otimizado para a produtividade sustentável e diversão do programador. Ele permite que você escreva código de maneira elegante, favorecendo convenção ao invés de configuração.”.

O *framework* surgiu em 2004 e foi criado por David Hanson, e utiliza linguagem *Ruby*, por isso seu nome *Ruby on Rails*.

Rails surgiu da união de alguns outros *frameworks* ganhando assim o nome de *meta-framework*, entre os *frameworks* que compõe o Rails estão:

- **Active Record:** Em Rails, é considerado um *framework* que contém uma camada de mapeamento objeto-relacional, entre a aplicação e o banco de dados;
- **Action Pack:** *framework* HTML, XML, Javascripts controle de regras de negócio;
- **Action Mailer:** *framework* recebimento de emails, capaz de realizar diversas operações apenas com chamadas de entregas de correspondência;
- **Active Support:** *framework* que contém coleções de diversas classes, e extensões de bibliotecas, consideradas úteis para uma aplicação em *Ruby On Rails*;
- **Active WebServices:** *framework* que provê uma maneira de publicar APIs que se comuniquem com o Rails.

O design no *framework* foi baseado em alguns conceitos chave: DRY (*Don't repeat yourself*) e o COC (*Convention over Configuration*) esse são dois dos principais conceitos que fazem o Rails se tornar realmente ágil. (MONTEIRO, 2012)

O DRY significa ao programador que ele não deve se repetir na codificação e sim concentrar o código em apenas um lugar e apenas realizar a chamada do mesmo no restante da aplicação. (MONTEIRO, 2012)

Para tornar o desenvolvimento inicial mais rápido o Rails faz uma série de suposições sobre o que o desenvolvedor vai usar, tornando praticamente inexistente configuração em arquivo XML ou qualquer outro tipo de arquivo, ao criar um projeto o desenvolvedor já por ir direto para a codificação. (THOMAS, 2008)

O Rails trabalha com a estrutura MVC e leva isso ao extremo deixando o código da aplicação completamente separado em seu devido lugar, mas ao mesmo tempo proporciona uma fácil interação entre modelo, visão e controlador.

Os papéis de cada dentro do MVC do Rails são divididos da seguinte maneira:

- **Controllers:** São responsáveis por atender todas as requisições e enviar as devidas respostas.
- **Models:** São responsáveis manter a comunicação entre a aplicação e o banco de dados, no modelo são onde são criadas validações de dados e a lógica de negocio da aplicação.
- **Views:** São a partes visuais da aplicação e geram arquivos em formato HTML, XML ou javascript.

Na Figura 3 é possível visualizar como o Rails trabalha o MVC:

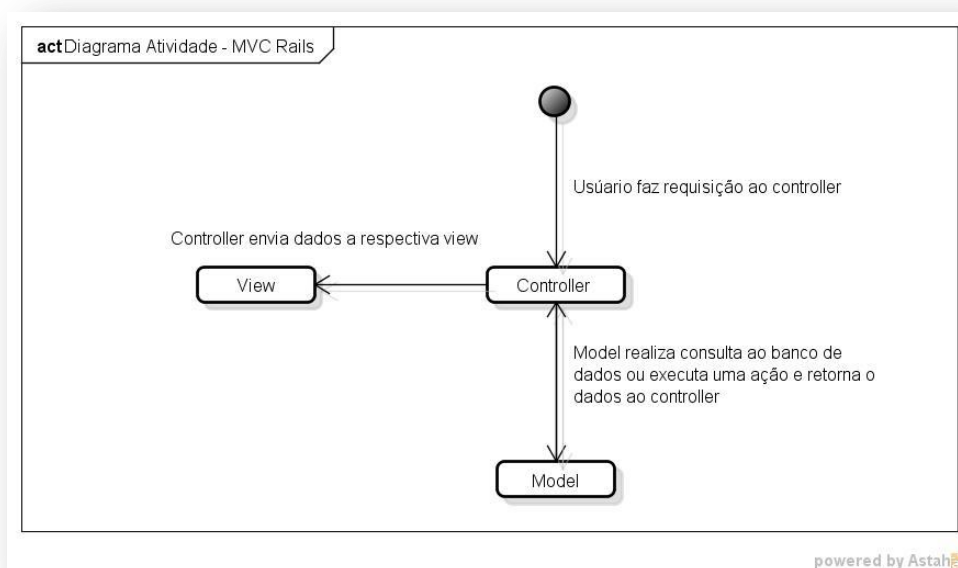


Figura 3 - Como funciona o MCV do Rails

Fonte: Autoria Própria

O usuário faz uma chamada a um *controller*, esse busca dados e verifica a lógica de negocio da aplicação em um *model* específico e retornar dados ao *controller*, o *controller* irá receber esses dados e enviá-los a *view* correspondente a requisição feita. (THOMAS, 2008)

Na Figura 4 é possível visualizar a estrutura de diretórios criada em um projeto Rails:

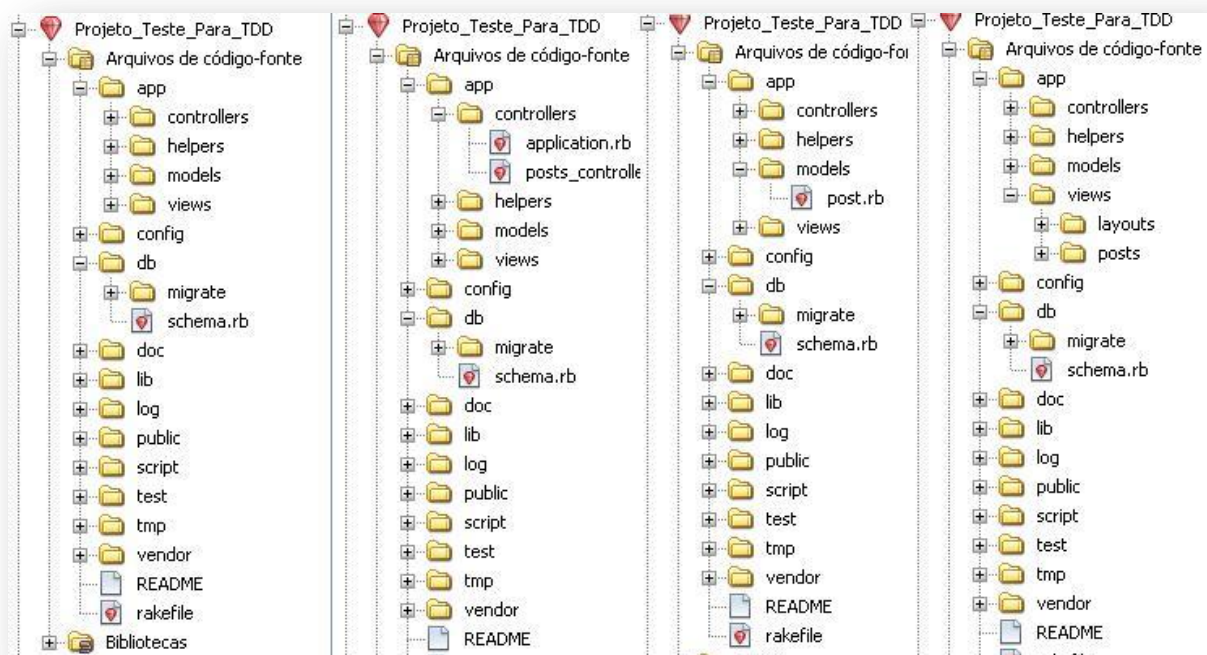


Figura 4 - Estrutura de uma aplicação Rails

Fonte: Autoria Própria

O Rails gera uma estrutura com locais destinados aos *controllers*, *model*, *views*, arquivos de javascript e diretório para testes. (MONTEIRO, 2012)

Os principais diretórios do Rails são descritos abaixo:

- **APP:** Se subdivide em outros diretórios e é onde o programador irá passar a maior parte do dentro escrevendo sua aplicação.
- **APP / controllers:** É o diretório responsável por abrigar os controladores da aplicação, que são as classes responsáveis por atender uma requisição e gerar uma resposta.
- **APP / models:** É o diretório responsável por abrigar os modelos da aplicação que são as classes responsáveis por interagir com banco de dados e conter a lógica de negocio.

- **APP / helpers:** É o diretório responsável por abrigar arquivos que contem métodos para facilitar a implementação de lógicas a serem mostradas na *views*.
- **APP / views:** É o diretório responsável por abrigar a telas da aplicação
- **CONFIG:** É o diretório responsável por abrigar arquivos de configuração da aplicação como arquivos de configuração de banco de dados e internacionalização.
- **DB / migrations:** É o diretório responsável por armazenar os arquivos de migrations que são arquivos responsáveis por gerar as tabelas do banco de dados a partir dos models da aplicação.
- **Public /** É o diretório onde ficam arquivos públicos da aplicação como arquivos de javascript, CSS, páginas estáticas e imagens.
- **Public / Images:** É o diretório responsável por armazenar as imagens da aplicação.
- **Public / stylesheet:** É o diretório responsável por abrigar os arquivos de CSS na aplicação.
- **Public / javascripts:** É o diretório responsável por armazenar os arquivos javascript na aplicação.

Além desses, o framework também cria diretórios destinados à inserção de plugins de terceiros, diretórios para armazenar comandos próprios e diretórios para construção dos testes automatizados da aplicação. (THOMAS, 2008)

2.8 TESTE COM FRAMEWORK RAILS

Escrever testes para aplicações Rails é simples, pois o próprio *framework* se encarrega de criar os arquivos de testes básicos à medida que modelos e controladores vão sendo gerados na sua aplicação. (SEA, 2009)

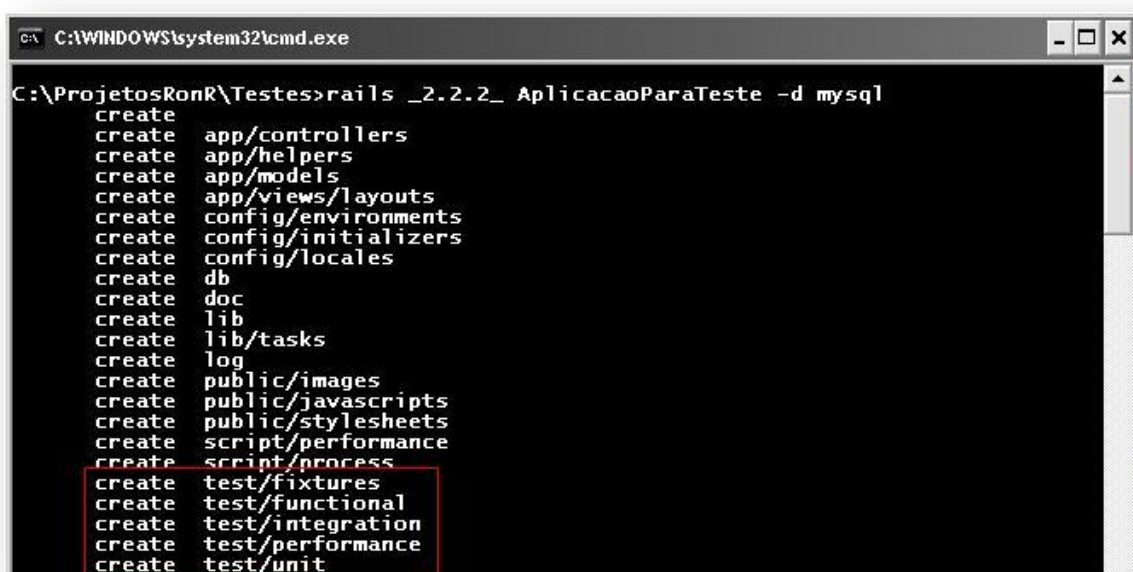
É possível aplicar testes que visam cobrir todas as áreas das aplicações, possibilitando testar entrada de dados, requisição e resposta dos controladores e fluxo da aplicação. (Guia do Rails., 201-)

Os Testes possíveis em uma aplicação usando Rails são:

- Testes Unitários: são testes aplicados aos modelos das aplicações Rails onde é possível testar métodos e entrada de dados. (THOMAS, 2008)
- Testes Funcionais: são testes aplicados aos controladores das aplicações Rails onde é possível simular requisições e validar as respostas geradas a cada ação do controlador. (THOMAS, 2008)
- Teste de Integração: são testes onde é possível testar um fluxo real da aplicação e interagir vários modelos e controladores ao mesmo tempo a fim de validar a fluxo da aplicação. (THOMAS, 2008)

Os testes possíveis serão explorados no decorrer do trabalho nas seções 2.8.4, 2.8.5 e 2.8.6.

Ao criar a aplicações o Rails se encarrega de criar os diretórios e arquivos bases para serem efetuados os testes, como pode ser visualizado na Figura 5.



```
C:\WINDOWS\system32\cmd.exe

C:\ProjetosRonR\Testes>rails _2.2.2_ AplicacaoParaTeste -d mysql
create
create  app/controllers
create  app/helpers
create  app/models
create  app/views/layouts
create  config/environments
create  config/initializers
create  config/locales
create  db
create  doc
create  lib
create  lib/tasks
create  log
create  public/images
create  public/javascripts
create  public/stylesheets
create  script/performance
create  script/process
create  test/fixtures
create  test/functional
create  test/integration
create  test/performance
create  test/unit
```

Figura 5 - Criação de Aplicação Rails e Diretórios para Testes

Fonte: Autoria Própria

O comando `rails _2.2.2_ AplicacaoParaTeste -d mysql` é responsável por criar a aplicação e é possível ver a criação de todos os diretórios com destaque aos diretórios de testes.

2.8.1 OS TRÊS AMBIENTES

Quando o projeto é criado, três bancos de dados são gerados, o banco de *development* que é o banco usado quando a aplicação está na fase de desenvolvimento, o banco de *production* que é o banco usando quando a aplicação está em produção e o banco *test* que é o banco utilizado para realizar os testes. Na Figura 6 é possível visualizar o arquivo responsável pela criação e configuração dos três bancos de dados. (MONTEIRO, 2012)

```
1 development:
2   adapter: mysql
3   encoding: utf8
4   database: AplicacaoParaTeste_development
5   pool: 5
6   username: root
7   password:
8   host: localhost
9
10 test:
11   adapter: mysql
12   encoding: utf8
13   database: AplicacaoParaTeste_test
14   pool: 5
15   username: root
16   password:
17   host: localhost
18
19 production:
20   adapter: mysql
21   encoding: utf8
22   database: AplicacaoParaTeste_production
23   pool: 5
24   username: root
25   password:
26   host: localhost
```

Figura 6 - Arquivo database.yml

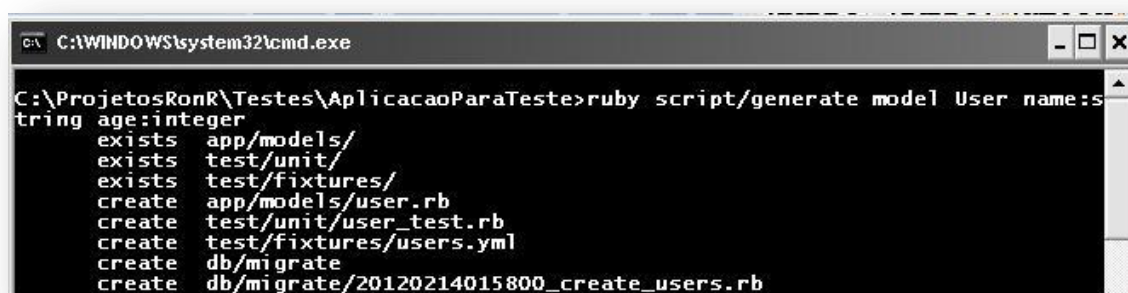
Fonte: Autoria Própria

2.8.2 FIXTURES

Fixtures são arquivos no formato .yml que permitem criar dados pré-definidos para que os mesmos possam ser carregados e utilizados para testes, são arquivos

que funcionam independentemente do banco de dados da aplicação. (MONTEIRO, 2012)

Os arquivos são encontrados no diretório test/fixtures e a cada *model* criado na aplicação um arquivo de fixtures é criado para o mesmo, isso pode ser visualizado na Figura 7.



```

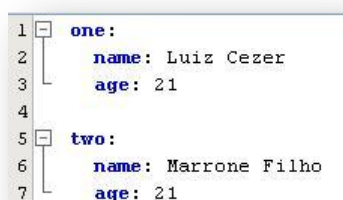
C:\WINDOWS\system32\cmd.exe

C:\ProjetosRonR\Testes\AplicacaoParaTeste>ruby script/generate model User name:string age:integer
exists    app/models/
exists    test/unit/
exists    test/fixtures/
create    app/models/user.rb
create    test/unit/user_test.rb
create    test/fixtures/users.yml
create    db/migrate
create    db/migrate/20120214015800_create_users.rb
  
```

Figura 7 - Criação de um Model e arquivos de fixtures na aplicação

Fonte: Autoria Própria

O comando `ruby script/generate model User` gera um novo modelo dentro da aplicação e também criará os arquivos para teste juntamente com a fixtures para serem usadas nos testes.



```

1 one:
2   name: Luiz Cezer
3   age: 21
4
5 two:
6   name: Marrone Filho
7   age: 21
  
```

Figura 8 - Fixture com dados para testes

Fonte: Autoria Própria

Na Figura 8 é apresentado o arquivo de fixture referente o modelo criado e com dados para ser utilizados durante os testes. Com os dados criados para simulação os mesmos podem ser facilmente utilizando nos testes e inseridos no banco de dados de teste.

2.8.3 ASSERTS

Segundo (Guia do Rails., 201-) “*Assertions* são as ‘formiguinhas’ dos testes. Elas são quem realmente realizam as checagens e garantem que as coisas estão correndo como o planejado.”. Segundo (THOMAS, 2008) “Uma assertiva é simplesmente uma chamada de método que informa ao *framework* o que esperamos ser verdade”.

Se nada errado acontecer à assertiva simplesmente retorna um valor *true*, mas caso o argumento passado seja *false* a assertiva falhará e o *framework* irá informar o erro (THOMAS, 2008).

2.8.4 TESTES UNITÁRIOS

Testes unitários são definidos como testes que visam testar pequenas unidades do projeto, sejam métodos do próprio *model* ou fazer validação de entrada de dados para o *model*. (THOMAS, 2008)

Os testes unitários são criados automaticamente cada vez que um novo *model* é criado na aplicação. Na Figura 9 é possível ver um exemplo de classe de teste unitário com um método padrão.

```
1  require 'test_helper'
2
3  class UserTest < ActiveSupport::TestCase
4    # Replace this with your real tests.
5    test "the truth" do
6      assert true
7    end
8  end
9  |
```

Figura 9 - Exemplo de classe de teste unitário

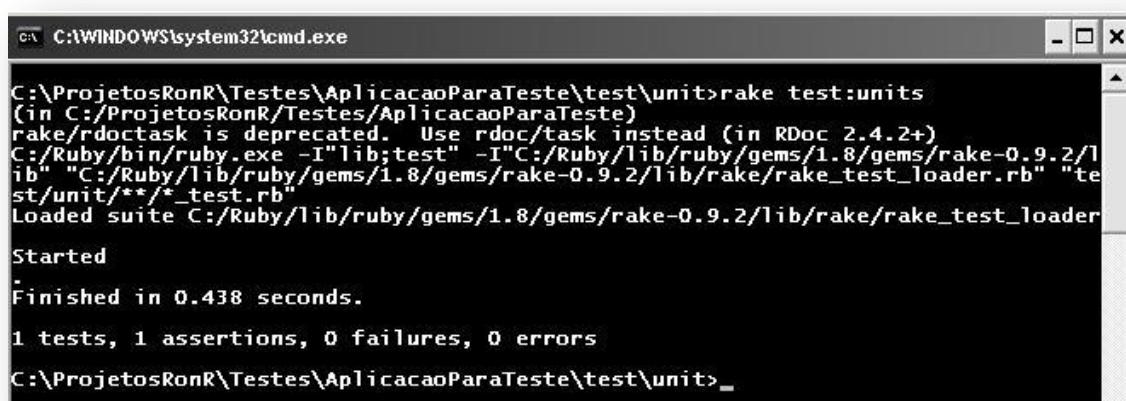
Fonte: Autoria Própria

Na primeira linha é possível observar que a classe faz um `require` ao arquivo `test_helper`, que por sua vez herda da classe `Unit::TestCase` que

assim como a classe `ActiveSupport::TestCase` são *frameworks* que possibilitam a criação e execução de métodos de teste.

Novos testes são definidos com a palavra *test* seguida do nome do teste a ser criado. Dentro dos testes são definidos *asserts*, uma *assert* é o método responsável por fazer as verificações dentro dos testes, um teste pode conter vários *asserts*.

O comando `rake test:units` executa os teste unitários e exibe os resultados, apontando se os testes foram bem sucedidos ou se houveram falhar e erros.



```

C:\WINDOWS\system32\cmd.exe

C:\ProjetosRonR\Testes\AplicacaoParaTeste\test\unit>rake test:units
(in C:/ProjetosRonR/Testes/AplicacaoParaTeste)
rake/rdoctask is deprecated. Use rdoc/task instead (in RDoc 2.4.2+)
C:/Ruby/bin/ruby.exe -I"lib:test" -I"C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib" "C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib/rake/rake_test_loader.rb" "test/unit/**/*.rb"
Loaded suite C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib/rake/rake_test_loader

Started
.
Finished in 0.438 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
C:\ProjetosRonR\Testes\AplicacaoParaTeste\test\unit>_
  
```

Figura 10 - Exibindo execução do teste

Fonte: Autoria Própria

Na Figura 10 é possível visualizar que o comando executa os testes presentes no arquivo exibindo seus resultados. O “.” significa que o teste passou, quando os testes falham para cada um é exibido um “F” e quando ocorrem erros que não possibilitaram a execução do teste é exibido um “E” (Guia do Rails., 201-).

2.8.5 TESTES FUNCIONAIS

Testes funcionais são testes que visam fazer verificações no *controllers* a fim de verificar a integridade das requisições, seja pra testar se a mesma foi feita

usando o método correto, se teve o retorno de dado esperado ou se foi redirecionada corretamente. (THOMAS, 2008)

Ao realizar testes funcionais o *framework* de teste disponibiliza cinco métodos de requisição e resposta são eles, *get, post, update, delete, xhr* e quatro *hashes* são eles *session, flash, cookies, assigns*. (SEA, 2009) .

Os testes funcionais são definidos pela palavra *test* seguida do nome do teste, dentro dele além dos métodos de requisição e repostas é possível usar assertivas para verificação dos dados testados. (Guia do Rails., 201-)

```
1 | require 'test_helper'
2
3 | class PostsControllerTest < ActionController::TestCase
4 |   test "should get index" do
5 |     get :index
6 |     assert_response :success
7 |     assert_not_nil assigns(:posts)
8 |   end
9 | end
```

Figura 11 - Exemplo de Teste Funcional

Fonte: Autoria Própria

Na Figura 11 tem-se um trecho de código de uma classe responsável pelos testes funcionais do controlador de *posts* de uma aplicação.

É possível observar o uso do método *get* na *action* *index* do controlador indicando que haverá uma simulação de chamada por *get* no método, além disso, existem assertivas para verificação de dados.

O comando `rake test:functionals` executa os testes funcionais um a um e usa a mesma analogia dos testes unitários para dizer se houveram erros, falhas ou se os testes foram bem executados como é visualizado na Figura 12

```

C:\> C:\WINDOWS\system32\cmd.exe

C:\ProjetosRonR\Testes\TesteParaTDD> rake test:functionals
rake/rdoctask is deprecated. Use rdoc/task instead (in RDoc 2.4.2+)
C:/Ruby/bin/ruby.exe -I"lib:test" -I"C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib" "C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib/rake/rake_test_loader.rb" "test/functional/**/*.test.rb"
Loaded suite C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib/rake/rake_test_loader
Started
.....
Finished in 0.594 seconds.

7 tests, 10 assertions, 0 failures, 0 errors
C:\ProjetosRonR\Testes\TesteParaTDD> _

```

Figura 12 - Executando os testes funcionais

Fonte: Autoria Própria

Segundo (THOMAS, 2008) “Ao testar controladores, estamos certificando de que uma dada requisição seja devolvida com uma resposta apropriada.”.

2.8.6 TESTE DE INTEGRAÇÃO

Testes de integração são testes utilizados para testar a interação entre vários *controllers* simultaneamente, em geral são utilizados para testar o fluxo dentro da aplicação e as respostas geradas por cada interação. (THOMAS, 2008)

Diferente dos outros testes os testes de integração devem ser explicitamente criados e ficam no diretório *test/integration*. (Guia do Rails., 201-)

```

C:\> C:\WINDOWS\system32\cmd.exe

C:\ProjetosRonR\Testes\AplicacaoParaTeste> ruby script/generate integration_test user_integration
exists test/integration/
create test/integration/user_integration_test.rb

C:\ProjetosRonR\Testes\AplicacaoParaTeste> _

```

Figura 13 - Criando um test integration

Fonte: Autoria Própria

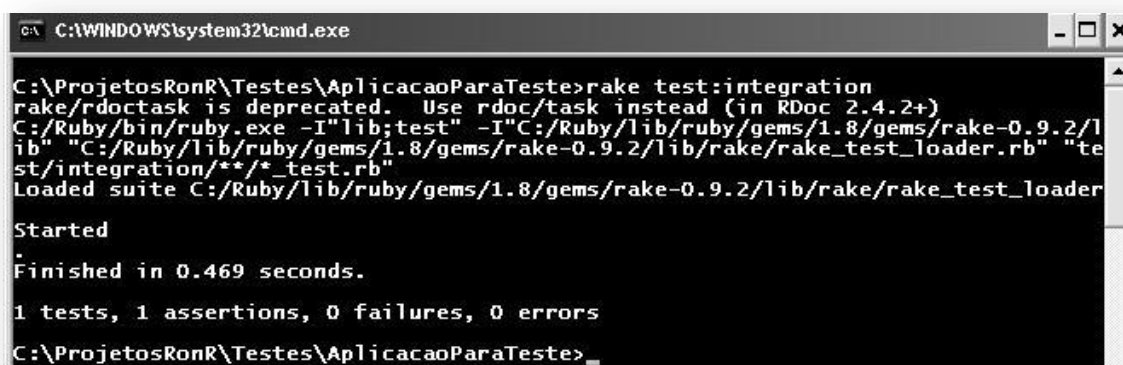
Na Figura 13 é possível visualizar o comando `script/generate integration_test user_integration` que é o responsável por criar o teste de integração.

```
1  require 'test_helper'
2
3  class UserIntegrationTest < ActionController::IntegrationTest
4    fixtures :all
5
6    # Replace this with your real tests.
7    test "the truth" do
8      assert true
9    end
10 end
```

Figura 14 - Exemplo de teste de integração

Fonte: Autoria Própria

Na Figura 14 é visualizado o exemplo de classe para teste de integração, na linha quatro existe o comando `fixtures:all` que quer dizer que a classe vai carregar todos os dados de *fixtures*, como esses testes visam verificar o fluxo da aplicação é possível que o mesmo teste contenha dados de vários *models*, então é preciso informar quais *models* o teste vai utilizar.



```
C:\WINDOWS\system32\cmd.exe
C:\ProjetosRonR\Testes\AplicacaoParaTeste>rake test:integration
rake/rdoctask is deprecated. Use rdoc/task instead (in RDoc 2.4.2+)
C:/Ruby/bin/ruby.exe -I"lib:test" -I"C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib" "C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib/rake/rake_test_loader.rb" "test/integration/*/*_test.rb"
Loaded suite C:/Ruby/lib/ruby/gems/1.8/gems/rake-0.9.2/lib/rake/rake_test_loader
Started
.
Finished in 0.469 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
C:\ProjetosRonR\Testes\AplicacaoParaTeste>_
```

Figura 15 - Executando teste de integração

Fonte: Autoria Própria

Na Figura 15 é possível visualizar a saída da execução do teste de integração.

3 MATERIAL E MÉTODOS

Esta seção tem como objetivo descrever as ferramentas e tecnologias utilizadas no desenvolvimento da aplicação experimental.

3.1 FERRAMENTAS UTILIZADAS

Para o desenvolvimento da aplicação experimental foram utilizadas as seguintes ferramentas e tecnologias:

- Linguagem Ruby.
- *Framework Rails*.
- Sistema Operacional Ubuntu
- Editor de texto gEdit
- Sistema Gerenciador de Banco de Dados MySQL
- Astah Community
- Servidor Web Webrick

O sistema operacional Ubuntu foi escolhido, pois permite um modo mais fácil de controlar as versões instaladas da linguagem Ruby e do *framework Rails* por meio de execuções de comandos no terminal.

O editor de texto gEdit é um editor padrão do Ubuntu e com a instalação de alguns *plugins* novos recursos ficam disponíveis tornando mais fácil o desenvolvimento com *framework Rails* sem a necessidade de um Ambiente de Desenvolvimento Integrado.

A linguagem Ruby usada no desenvolvimento da aplicação experimental está na versão 1.9.2 sendo uma das últimas versões lançada, sendo assim uma versão atual, assim com a versão do *framework Rails* que está na versão 3.0.4.

O *framework Rails* pode conectar-se facilmente a diversos sistemas de banco de dados, o escolhido para o desenvolvimento foi o MySQL sendo um banco de fácil instalação de manutenção no Ubuntu.

O servidor web responsável por executar a aplicação foi o servidor Webrick, servidor que é instalado juntamente quando o Ruby é instalado.

A ferramenta Astah Community foi utilizada como ferramenta para modelagem dos diagramas UML apresentados no trabalho.

3.2 ESTUDO EXPERIMENTAL

Na seção será descrito o procedimento de desenvolvimento da aplicação para estudo experimental sobre desenvolvimento guiado a testes com *framework Rails*.

A aplicação a ser desenvolvida se baseia em uma lista de tarefas, onde um usuário autenticado no sistema poderá criar suas novas tarefas a serem feitas. Na Figura 16 é visualizado o diagrama de caso de uso da aplicação, com as principais funcionalidades. Um usuário após realizar a ação de efetuar *login*, poderá realizar as tarefas de gerenciamento de uma nova tarefa, quando não está autenticado pode apenas visualizar os detalhes de uma tarefa.

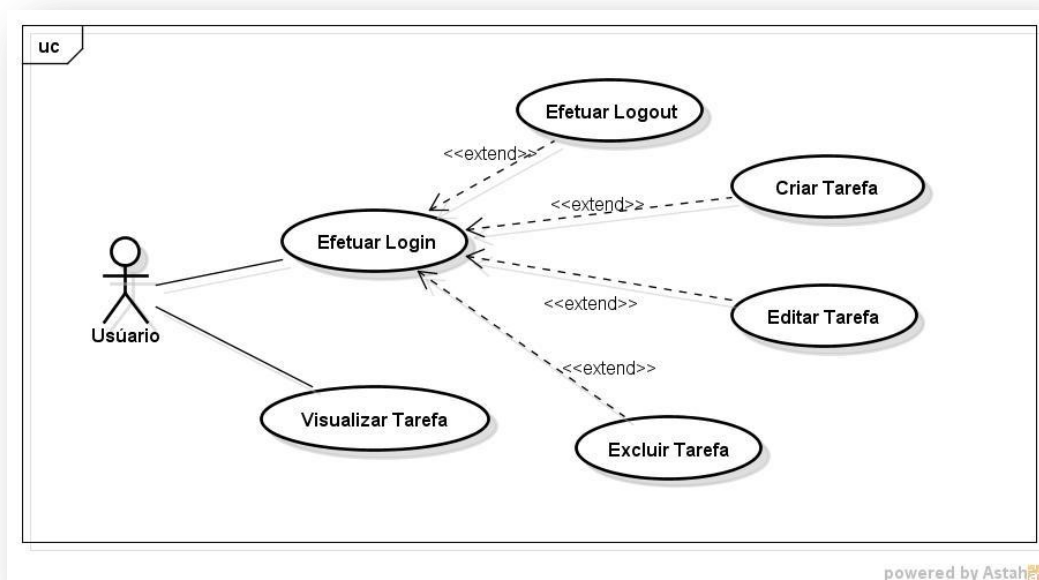


Figura 16 - Caso de Uso aplicação

Fonte: Autoria Própria

3.2.1 Criando aplicação e principais recursos

Para criação de aplicação de lista de tarefas foi necessário à criação de dois recursos, um recurso *model User* que será o responsável por gerenciar os usuários de aplicação, o recurso *Task* que irá ser a nova tarefa a ser gerenciada.



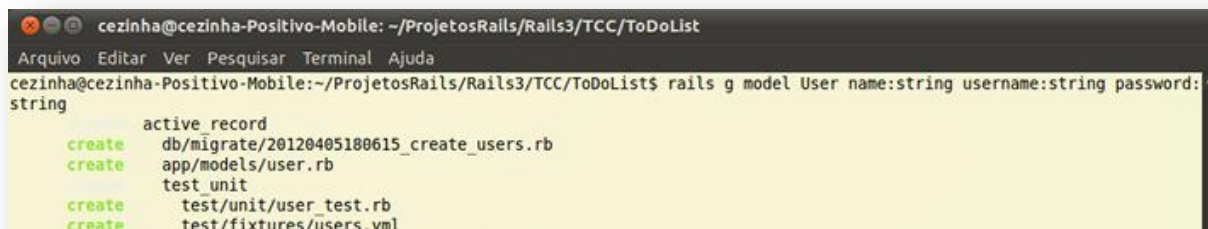
```
cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC
Arquivo Editar Ver Pesquisar Terminal Ajuda
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC$ rails new ToDoList -d mysql
create
create  README
create  Rakefile
create  config.ru
create  .gitignore
create  Gemfile
create  app
```

Figura 17 - Criando Aplicação

Fonte: Autoria Própria

Na Figura 17 por meio do comando: `rails new ToDoList -d mysql`, será criada uma nova aplicação Rails com utilização do banco de dados MySQL.

Com a aplicação criada é necessário agora criar o recurso de User para manutenção de usuários.



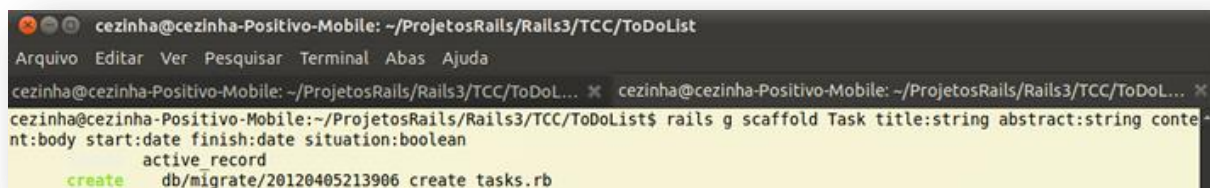
```
cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Ajuda
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$ rails g model User name:string username:string password:string
create  active_record
create  db/migrate/20120405180615_create_users.rb
create  app/models/user.rb
create  test_unit
create  test/unit/user_test.rb
create  test/fixtures/users.yml
```

Figura 18 - Criação de Model User

Fonte: Autoria Própria

A Figura 18 mostra a criação do *model* por meio do comando `rails g model User`. A execução deste comando irá gerar um modelo para aplicação, já com os atributos *name*, *username* e *password*.

Na Figura 19 é realizada a criação de um *scaffold*¹ que irá gerar o *controller*, o *model* e as *views* para o recurso *Task*.



```

cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Abas Ajuda
cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList$ rails g scaffold Task title:string abstract:string content:string body:string start:date finish:date situation:boolean active_record
create db/migrate/20120405213906_create_tasks.rb

```

Figura 19 - Criando Scaffold Task

Fonte: Autoria Própria

3.2.2 Criando testes unitários para model User

Quando um *model* é gerado para uma aplicação *Rails* ele automaticamente gera seus arquivos de testes e é neles onde os testes serão escritos.

Na Figura 20 tem-se o arquivo onde os testes foram escritos. O primeiro teste que começa na linha 6 e termina na linha 9 visa validar que o campo *name* não seja gravado de maneira nula pelo sistema de permitindo que uma mensagem de erro seja enviada caso o teste falhe. O segundo teste que inicia na linha 11 e termina na linha 14 é o teste que visa validar para que o campo *username* também não seja nulo e dispare um erro caso o teste falhe, de mesmo modo, o terceiro teste que inicia na linha 16 e termina na linha 19 visa garantir que o campo *password* não seja nulo, disparando uma mensagem caso o teste falhe.

¹ Segundo (Guia do Rails., 201-) “O *scaffolding* do Rails é um modo rápido para gerar algumas das partes principais de uma aplicação. Se for necessário criar os *models*, *views* e *controllers* para um novo recurso em uma única operação, o *scaffolding* é a ferramenta para o trabalho.”.

```
1 require 'test_helper'
2
3 class UserTest < ActiveSupport::TestCase
4   # Replace this with your real tests.
5
6   test 'name can not be nil' do
7     user = create(:name => nil)
8     assert !user.save, 'name can not be nil'
9   end
10
11   test 'username can not be nil' do
12     user = create(:username => nil)
13     assert !user.save, 'username can not be nil'
14   end
15
16   test 'password can not be nil' do
17     user = create(:password => nil)
18     assert !user.save, 'password can not be nil'
19   end
20
21   private
22   def create(options = {})
23     User.create({
24       :name => 'Luiz',
25       :username => 'luizcezer',
26       :password => '123456',
27     }.merge(options))
28   end
29 end
30
31
```

Figura 20 - Teste de validação de dados model User

Fonte: Autoria Própria

Entre as linhas 23 e 29 tem-se a criação de um objeto padrão que será passado aos métodos, caso esse objeto não tenha nenhum argumento, ele será criado com os dados pré-definidos.

Como o TDD adota a técnica de testar antes e ver os testes falhando é preciso rodar os testes acima e vê-los falhar, o comando `rake test:units` é o responsável por executar os testes e gerar a saída que pode ser visualizada na Figura 21


```

cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Ajuda
... Rake::DSL into classes and modules which use the Rake DSL methods.
WARNING: DSL method ToDoList::Application#task called at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/railties-3.0.4/
s/application.rb:214:in `initialize_tasks'
Loaded suite /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rake_test_loader
Started
FFF
Finished in 0.234866 seconds.

1) Failure:
test_name_can not be nil(UserTest) [/home/cezinha/ProjetosRails/Rails3/TCC/ToDoList/test/unit/user_test.rb:8]:
name can not be nil

2) Failure:
test_password_can not be nil(UserTest) [/home/cezinha/ProjetosRails/Rails3/TCC/ToDoList/test/unit/user_test.rb:18]:
password can not be nil

3) Failure:
test_username_can not be nil(UserTest) [/home/cezinha/ProjetosRails/Rails3/TCC/ToDoList/test/unit/user_test.rb:13]:
username can not be nil

3 tests, 3 assertions, 3 failures, 0 errors, 0 skips

```

Figura 21 - Teste falhando

Fonte: Autoria Própria

A saída no console indica que houveram três erros nos testes executados, o primeiro erro no teste de validação de nome indica que o campo foi informado como nulo, o mesmo acontece para os outros testes informado o mesmo erro.

Seguindo a filosofia do TDD o segundo passo é escrever o mínimo de código possível para que o teste passe.

```

1 class User < ActiveRecord::Base
2   validates :name, :presence => true
3   validates :username, :presence => true
4   validates :password, :presence => true
5 end
6

```

Figura 22 - Mínimo de código para o teste passar

Fonte: Autoria Própria

Na Figura 22 é codificado o suficiente de código para que o teste passe. sendo criadas validações de presença para os campos *name*, *username* e

password, ou seja, os campos agora precisam ser preenchidos caso contrário um erro de validação será disparado.

Agora rodando o teste novamente, tem-se uma nova saída que pode ser visualizada na Figura 23.



```

cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Ajuda
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$ rake test:units
WARNING: 'require 'rake/rdoctask'' is deprecated. Please use 'require 'rdoc/task' (in RDoc 2.4.2+)' instead.
  at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rdoctask.rb
WARNING: Global access to Rake DSL methods is deprecated. Please include
  ... Rake::DSL into classes and modules which use the Rake DSL methods.
WARNING: DSL method ToDoList::Application#task called at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/railties-
s/application.rb:214:in 'initialize tasks'
Loaded suite /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rake_test_loader
Started
...
Finished in 0.211714 seconds.

3 tests, 3 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 23180
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$

```

Figura 23 - Testes sendo executados sem erros

Fonte: Autoria Própria

A saída agora mudou e os testes passaram, garantindo assim que a validação feita no modelo para que os campos fossem preenchidos funcione de maneira adequada, não permitindo assim ao cadastrar um usuário com um campo inválido ou nulo.

Outra validação que precisa ser feita é a validação para que o campo *password* tenha o mínimo de seis caracteres. Primeiro cria-se o teste para fazer essa validação que pode ser visualizado na Figura 24.

```

21 test 'password can not be short' do
22   user = create(:password => '12345')
23   assert user.save, 'password can not be short, minimun 6 characters'
24 end
25

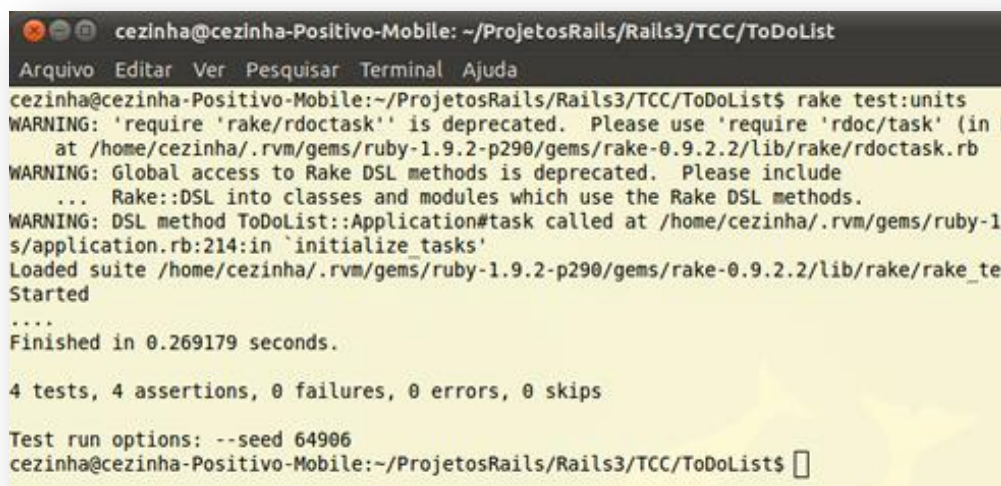
```

Figura 24 - Teste de validação de número de caracteres

Fonte: Autoria Própria

O teste criado valida para que o campo *password* tenha no mínimo seis caracteres e dispara um erro caso isso não ocorra.

Rodando o teste é possível observar na sua saída que nenhum erro aconteceu, o problema é que ainda não foi informado ao modelo que ele precisa aceitar um número mínimo de seis caracteres. Na Figura 25 é visualizada essa saída.



```

cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Ajuda
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$ rake test:units
WARNING: 'require 'rake/rdoctask'' is deprecated. Please use 'require 'rdoc/task' (in F
at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rdoctask.rb
WARNING: Global access to Rake DSL methods is deprecated. Please include
... Rake::DSL into classes and modules which use the Rake DSL methods.
WARNING: DSL method ToDoList::Application#task called at /home/cezinha/.rvm/gems/ruby-1.
s/application.rb:214:in `initialize_tasks'
Loaded suite /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rake_test
Started
....
Finished in 0.269179 seconds.

4 tests, 4 assertions, 0 failures, 0 errors, 0 skips

Test run options: --seed 64906
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$

```

Figura 25 - Teste de número de caracteres não dispara erro

Fonte: Autoria Própria

Como o *model* ainda não foi informado que só deve aceitar seis caracteres para o campo *password* nenhum erro é disparado, pois nenhuma validação precisou ser feita. Na Figura 26 é informado ao *model* que ele deve aceitar no mínimo seis caracteres para o campo *password*.

```

1 class User < ActiveRecord::Base
2   validates :name, :presence => true
3   validates :username, :presence => true
4   validates :password, :presence => true, :length => {:minimum => 6}
5 end
6

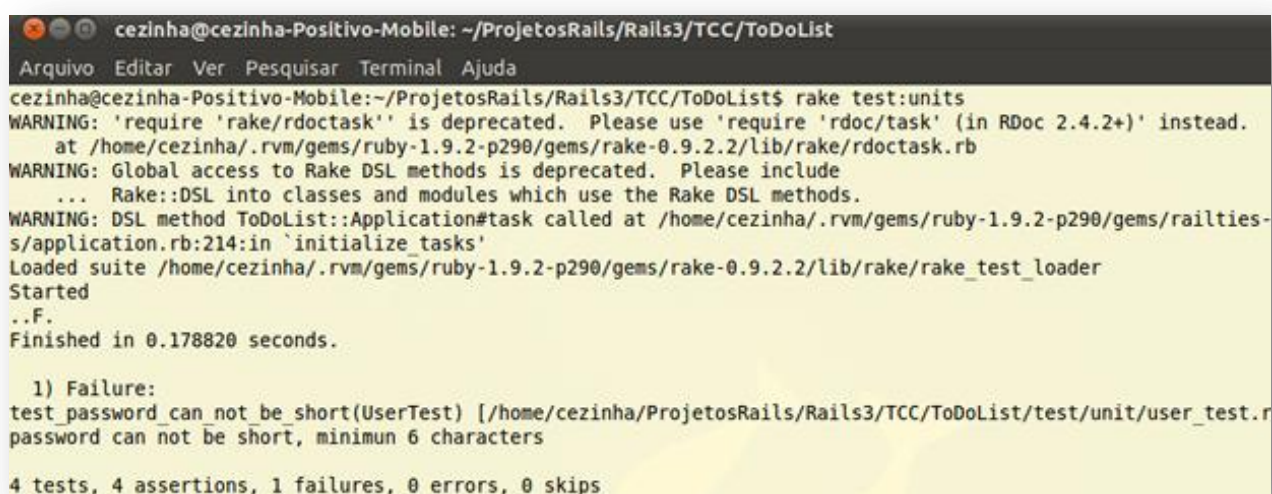
```

Figura 26 - Ajuste no model

Fonte: Autoria Própria

Na linha 4 é informado ao *model* que além de forçar que o campo *password* seja preenchido, também é validado para que o mesmo tenha no mínimo seis caracteres ao ser preenchido.

Agora rodando novamente os testes eles falham, pois o campo *password* no teste tem somente cinco caracteres que é validado pelo *model*, o teste não passa e a mensagem informada é vista no console dizendo que é preciso um mínimo de seis caracteres para que o campo *password* seja validado.



```

cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Ajuda
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$ rake test:units
WARNING: 'require 'rake/rdoctask'' is deprecated. Please use 'require 'rdoc/task' (in RDoc 2.4.2+)' instead.
  at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rdoctask.rb
WARNING: Global access to Rake DSL methods is deprecated. Please include
  ... Rake::DSL into classes and modules which use the Rake DSL methods.
WARNING: DSL method ToDoList::Application#task called at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/railties-
s/application.rb:214:in `initialize_tasks'
Loaded suite /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rake_test_loader
Started
..F.
Finished in 0.178820 seconds.

1) Failure:
test_password_can_not_be_short(UserTest) [/home/cezinha/ProjetosRails/Rails3/TCC/ToDoList/test/unit/user_test.r
password can not be short, minimum 6 characters

4 tests, 4 assertions, 1 failures, 0 errors, 0 skips

```

Figura 27 - Teste de número de caracteres falhando

Fonte: Autoria Própria

Para garantir que o teste esteja com seu comportamento funcionando adequadamente é preciso ajustá-lo para que ele tenha seis caracteres e dessa vez possa passar, fazendo assim a validação e o teste terem o comportamento esperado.

```

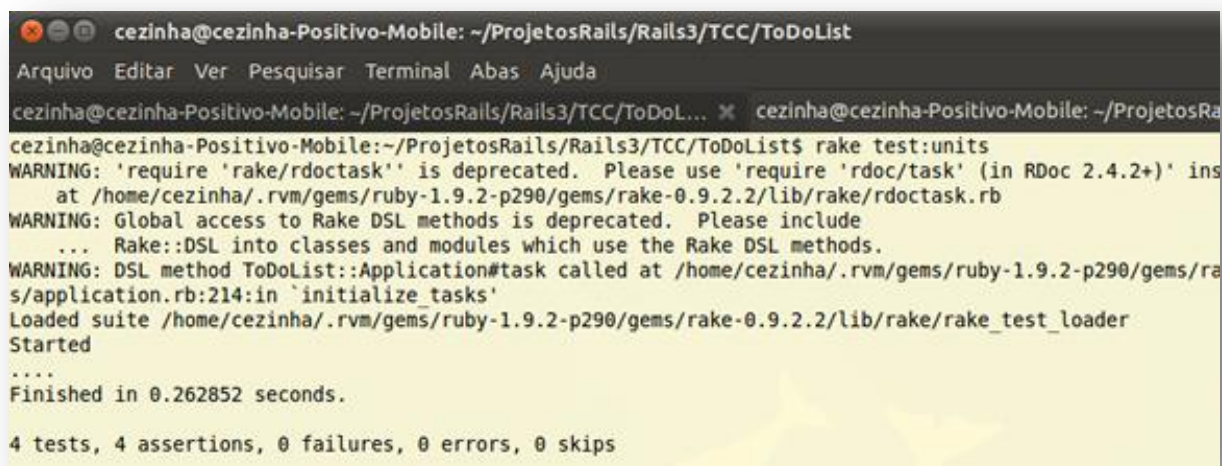
21 test 'password can not be short' do
22   user = create(:password => '123456')
23   assert user.save, 'password can not be short, minimum 6 characters'
24 end

```

Figura 28 - Ajustando teste de número de caracteres

Fonte: Autoria Própria

Na Figura 28 o teste é ajustado e, agora o campo tem seis caracteres como é o requerido pelo *model*. Na Figura 29 os testes são executados novamente e dessa vez todos passam dando a garantia assim que todas as validações e comportamentos desejados ao *model User* estão funcionando de maneira adequada.



```

cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Abas Ajuda
cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList$ rake test:units
WARNING: 'require 'rake/rdoctask'' is deprecated. Please use 'require 'rdoc/task' (in RDoc 2.4.2+) ' ins
  at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rdoctask.rb
WARNING: Global access to Rake DSL methods is deprecated. Please include
  ... Rake::DSL into classes and modules which use the Rake DSL methods.
WARNING: DSL method ToDoList::Application#task called at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/r
s/application.rb:214:in `initialize_tasks'
Loaded suite /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rake_test_loader
Started
....
Finished in 0.262852 seconds.

4 tests, 4 assertions, 0 failures, 0 errors, 0 skips

```

Figura 29 - Todos os testes passam para o model User

Fonte: Autoria Própria

3.2.3 Criação de teste funcional para *controller Task*

Como o recurso *Task* foi gerado utilizando o recurso de *scaffold* do *Rails*, além do *controller*, *model* e *views* referentes ao recurso *Task* também foram criados testes para todas essas partes.

Com a utilização do *scaffold*, um esqueleto com alguns testes já configurados é criado para o *controller Task* como mostra a Figura 30.

```

19 test "should get new" do
20   get :new
21   assert_response :success
22 end
23
24 test "should create task" do
25   assert_difference('Task.count') do
26     post :create, :task => @task.attributes
27   end
28
29   assert_redirected_to task_path(assigns(:task))
30 end
31
32
33
34 test "should get edit" do
35   get :edit, :id => @task.to_param
36   assert_response :success
37 end
38
39 test "should update task" do
40   put :update, :id => @task.to_param, :task => @task.attributes
41   assert_redirected_to task_path(assigns(:task))
42 end
43
44 test "should destroy task" do
45   assert_difference('Task.count', -1) do
46     delete :destroy, :id => @task.to_param
47   end
48
49   assert_redirected_to tasks_path
50 end

```

Figura 30 - Esqueleto de testes funcionais

Fonte: Autoria Própria

O teste que irá ser realizado visará garantir que somente quando o usuário estiver autenticado poderá ter acesso as ações de *edit*, *update*, *destroy*, *new* e *create* do *controller* de *Task*.

Para isso primeiro é preciso informar o *controller Task* que antes de chamar essas ações ele precisará chamar outra ação a fim de verificar se existe algum usuário autenticado na aplicação.

```

1 class TasksController < ApplicationController
2   # GET /tasks
3   # GET /tasks.xml
4
5   before_filter :authorize, :except => [:index, :show]
6

```

Figura 31 - Utilização de filtro no controller

Fonte: Autoria Própria

Na Figura 31 o método `before_filter` chamado na linha 5 é um método do *Rails* que irá informar ao *controller* que antes de qualquer ação primeiro ele deverá chamar o método `authorize` que irá verificar se há alguém autenticado na aplicação, caso não tenha ele irá bloquear o acesso a todas as ações, menos a ação de *index* e *show*.

O método `authorize` é um método criado no *controller* principal da aplicação e nele é feita a verificação se há ou não um usuário logado na aplicação, conforme a Figura 32.

```
4 def authorize
5   user = User.find(session[:user_id])
6   if user.nil?
7     flash[:notice] = 'Precisa logar'
8     redirect_to new_session_path
9   end
10 end
```

Figura 32 - Método authorize no controller principal

Fonte: Autoria Própria

A variável `session[:user_id]` é a variável que informa a aplicação se existe ou não um usuário autenticado, é criada no *model* de *User* assim que um usuário se autentica na aplicação.

```
6 def create
7   user = User.authenticate(params[:username],params[:password])
8   if user
9     session[:user_id] = user.id
10    flash[:notice] = 'Logado com sucesso'
11    redirect_to tasks_path
12  else
13    session[:user_id] = nil
14    flash[:notice] = 'Senha e/ou Password invalidos'
15    redirect_to tasks_path
16  end
17 end
18
19 def destroy
20   unless session[:user_id].nil?
21     session[:user_id] = nil
22     flash[:notice] = 'Desligado com sucesso!'
23     redirect_to tasks_path
24   end
25 end
```

Figura 33 - Ações responsáveis por login e logout de usuário

Fonte: Autoria Própria

Na Figura 33 são exibidos os métodos responsáveis pelas ações de *login* e *logout* de *User*, na ação *create* é chamado o método *authenticate* do *model User* e feita a verificação com os dados passados pelo usuário, se os dados forem encontrados no banco de dados, a autenticação é bem sucedida e a variável `session[user_id]` é criada, já a ação *destroy* é a responsável por efetuar o *logout* do usuário.

Como visualizado na Figura 33 o *model User* possui um método chamado *authenticate* que irá realizar a busca de usuário com os dados passados, esse método é visualizado conforme Figura 34.

```
6 def authenticate(username,password)
7   user = find_by_username(username)
8   if user
9     if user.password != password
10      user = nil
11    end
12  end
13  user
14 end
```

Figura 34 - Método authenticate

Fonte: Autoria Própria

Executando os testes funcionais do *controller* a saída gerada é visualizada na Figura 35.

```
7 tests, 3 assertions, 0 failures, 5 errors, 0 skips
Test run options: --seed 22808
rake aborted!
Command failed with status (1): [/home/cezinha/.rvm/rubies/ruby-1.9.2-p290/...]
Tasks: TOP => test:functionals
```

Figura 35 - Erros no teste funcional

Fonte: Autoria Própria

A saída gerada mostra que nem todos os testes e nem todas as asserções foram executadas, pois como o filtro foi inserido no *controller* os testes não

conseguiram ser executado da maneira esperada ocasionando o erro, o que mostra que a verificação do filtro está funcionando da maneira esperada.

Para verificar se o usuário este autenticado e tenha acesso a todas as ações do *controller* é preciso alterar o testes simulando uma autenticação para que todos os testes possam ser executados. Um método pode ser adicionado ao *test_helper* (Figura 36) a fim de simular um usuário autenticado na aplicação.

```
15 def login_as(user)
16   @request.session[:user_id] = users(user).id
17 end
```

Figura 36 - Método que simula autenticação de usuário

Fonte: Autoria Própria

O método *login_as* simula a existência um usuário autenticado na aplicação recuperando dados do arquivo de *fixture* de *User*.

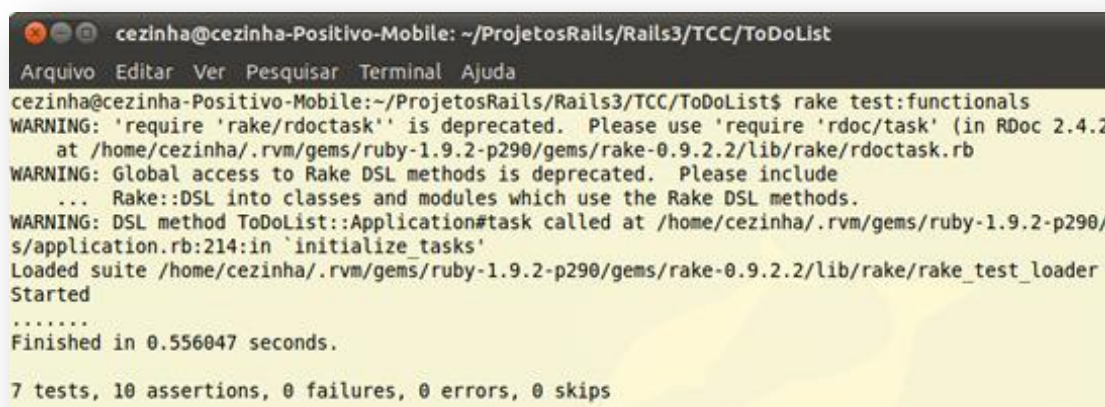
```
19 test "should get new" do
20   login_as(:userUm)
21   get :new
22   assert_response :success
23 end
24
25 test "should create task" do
26   login_as(:userUm)
27   assert_difference('Task.count') do
28     post :create, :task => @task.attributes
29   end
30   assert_redirected_to task_path(assigns(:task))
31 end
32
33 test "should get edit" do
34   login_as(:userUm)
35   get :edit, :id => @task.to_param
36   assert_response :success
37 end
38
39 test "should update task" do
40   login_as(:userUm)
41   put :update, :id => @task.to_param, :task => @task.attributes
42   assert_redirected_to task_path(assigns(:task))
43 end
44
45 test "should destroy task" do
46   login_as(:userUm)
47   assert_difference('Task.count', -1) do
48     delete :destroy, :id => @task.to_param
49   end
50   assert_redirected_to tasks_path
51 end
```

Figura 37 - Inserindo método login_as aos testes

Fonte: Autoria Própria

Com o método criado é preciso alterar o esqueleto de testes gerados e informar ao mesmo que antes de executar os testes é preciso chamar esse método e verificar que há um usuário autenticado.

Com a inserção do método *login_as* aos testes como mostra a Figura 37, é simulado que um usuário está autenticado na aplicação e assim ele poderá ter acesso aos métodos sem que nenhum erro seja disparado. Rodando novamente os testes a saída informa que todos eles passaram como visualizado na Figura 38



```

cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Ajuda
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$ rake test:functionals
WARNING: 'require 'rake/rdoctask'' is deprecated. Please use 'require 'rdoc/task' (in RDoc 2.4.2+)
at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rdoctask.rb
WARNING: Global access to Rake DSL methods is deprecated. Please include
... Rake::DSL into classes and modules which use the Rake DSL methods.
WARNING: DSL method ToDoList::Application#task called at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/application.rb:214:in 'initialize_tasks'
Loaded suite /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rake_test_loader
Started
.....
Finished in 0.556047 seconds.

7 tests, 10 assertions, 0 failures, 0 errors, 0 skips

```

Figura 38 - Teste funcionais funcionando

Fonte: Autoria Própria

Com a finalização da criação dos testes funcionais foi possível garantir que as ações *update*, *edit*, *create*, *new*, *destroy* do *controller Task* só pudessem ser acessadas se um usuário estivesse autenticado na aplicação caso contrário fica impossível seu acesso, garantindo assim o comportamento adequada da aplicação.

3.2.4 Teste de integração

Os testes de integração criados visam garantir que seja possível simular acesso a página de *login* e também simular que algum usuário faça seu login, também é preciso simular a requisição a um logout de usuário e por fim simular o fluxo entre todos os recursos onde um usuário faria login, criaria uma nova tarefa na aplicação e faria seu *logout*.

Os testes de integração não são criados automaticamente como os testes de unidade e funcionais, é preciso informar ao *Rails* que se deseja criar um novo teste de integração.

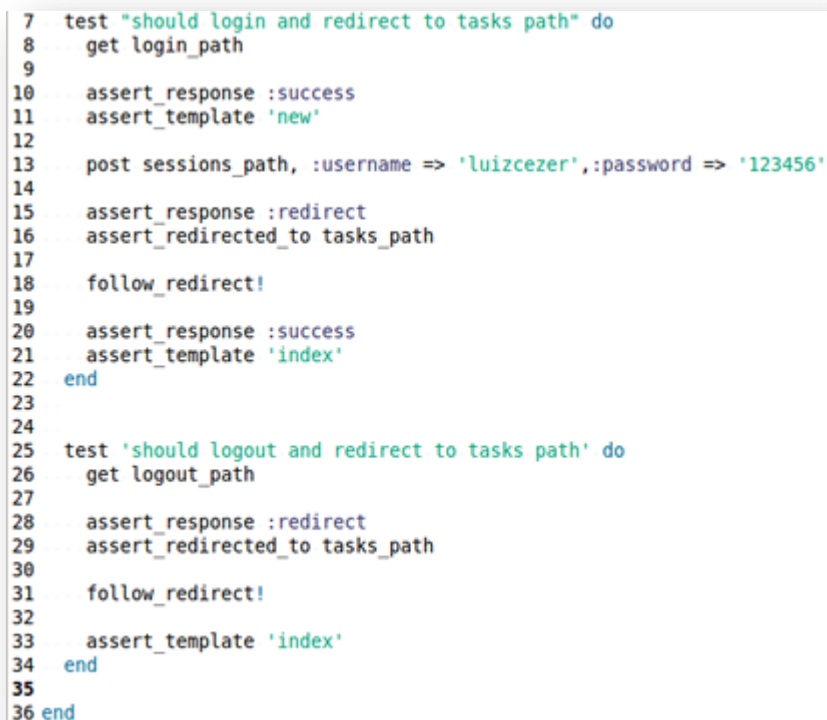


```
cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Ajuda
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$ rails g test_unit:integration UserStories
create test/integration/user_stories test.rb
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$
```

Figura 39 - Criando teste de integração

Fonte: Autoria Própria

Na Figura 39 por meio do comando `rails g test_unit:integration UserStories` é informado ao *Rails* para criar um novo teste de integração para a aplicação. Agora a aplicação já tem o arquivo necessário para criação do teste, primeiro serão criados os testes para simular *login* e *logout* de usuários na aplicação, conforme a Figura 40.



```
7 test "should login and redirect to tasks path" do
8   get login_path
9
10  assert_response :success
11  assert_template 'new'
12
13  post sessions_path, :username => 'luizcezer', :password => '123456'
14
15  assert_response :redirect
16  assert_redirected_to tasks_path
17
18  follow_redirect!
19
20  assert_response :success
21  assert_template 'index'
22 end
23
24
25 test 'should logout and redirect to tasks path' do
26   get logout_path
27
28   assert_response :redirect
29   assert_redirected_to tasks_path
30
31   follow_redirect!
32
33   assert_template 'index'
34 end
35
36 end
```

Figura 40 - Testes de integração para login e logout

Fonte: Autoria Própria

O primeiro teste visa simular um login de usuário, o primeiro passo é chamar a ação responsável para isto como visualizado na linha 8, depois disso são simulados respostas de sucesso e uma view é chamada como visualizado nas linhas 10 e 11. A linha 13 é a responsável por simular uma requisição *POST* para o caminho responsável por realizar a autenticação enviando dados de *username* e *password*. Na linha 15 é dito que a ação foi bem sucedida e é chamado um redirecionamento pela aplicação, esse redirecionamento é visualizado na linha 16 onde o fluxo é enviado para a ação responsável por exibir todas as tarefas.

O próximo teste irá garantir que o funcionamento da ação de *logout* está adequado, na linha 26 a ação de *logout* é chamada via *GET* e após o *logout* ser efetuado a aplicação chama um redirecionamento para a página onde se encontram as tarefas como é visualizado nas linhas 28 e 29, ao final na linha 33 é testado se ação irá retornar uma página chamada *index*.

Por fim ainda falta escrever o teste que irá simular um fluxo normal na aplicação aonde um usuário irá se autenticar criar uma nova tarefa e realizar o *logout*, esse teste é visualizado na Figura 41.

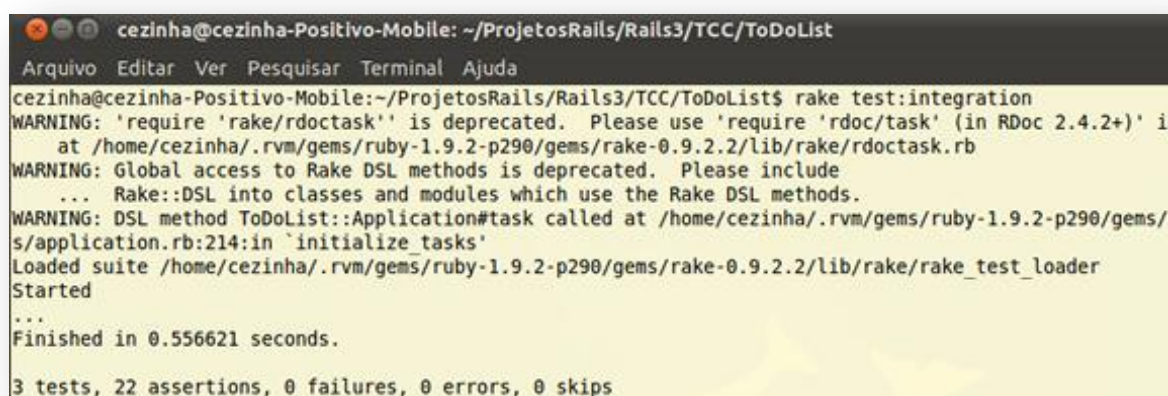
```
36 test 'should login create new task and logout' do
37   #login
38   get logout_path
39   assert_response :redirect
40   assert_redirected_to tasks_path
41   follow_redirect!
42   assert_template 'index'
43
44   #create new Task
45   get new_task_path
46   assert_response :success
47   assert_template 'new'
48   post tasks_path, :task => {:title => 'Primeira Tarefa', :abstract => 'devo
49   assert assigns(:task).valid?
50   assert_response :redirect
51   assert_redirected_to task_path(assigns(:task))
52   follow_redirect!
53   assert_response :success
54   assert_template 'show'
55
56   #logout
57   get logout_path
58   assert_response :redirect
59   assert_redirected_to tasks_path
60   follow_redirect!
61   assert_template 'index'
62 end
```

Figura 41 - Teste de fluxo normal da aplicação

Fonte: Autoria Própria

O teste acima irá simular um fluxo normal, após o usuário realizar seu login na aplicação ele irá solicitar a ação pra criar uma nova tarefa conforme a linha 45, após isso a aplicação irá retornar uma ação de sucesso e chamar a *view* correspondente, conforme linha 47, para criação de uma nova tarefa. Na linha 48 é simulada uma requisição *POST* enviados dados para criação de nova tarefa, a linha 49 verifica se o novo objeto é um objeto válido e após isso fará o redirecionamento para a página de *show*, conforme linha 54. Por fim a simulação de *logout* é realizada finalizando assim o teste de fluxo da aplicação.

Para garantir que os teste funcionam é preciso rodar o comando `rake test:integration`, conforme a Figura 42.



```
cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList
Arquivo Editar Ver Pesquisar Terminal Ajuda
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$ rake test:integration
WARNING: 'require 'rake/rdoctask'' is deprecated. Please use 'require 'rdoc/task' (in RDoc 2.4.2+)' in
at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rdoctask.rb
WARNING: Global access to Rake DSL methods is deprecated. Please include
... Rake::DSL into classes and modules which use the Rake DSL methods.
WARNING: DSL method ToDoList::Application#task called at /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/
s/application.rb:214:in 'initialize tasks'
Loaded suite /home/cezinha/.rvm/gems/ruby-1.9.2-p290/gems/rake-0.9.2.2/lib/rake/rake_test_loader
Started
...
Finished in 0.556621 seconds.

3 tests, 22 assertions, 0 failures, 0 errors, 0 skips
```

Figura 42 - Resultados para os testes de integração

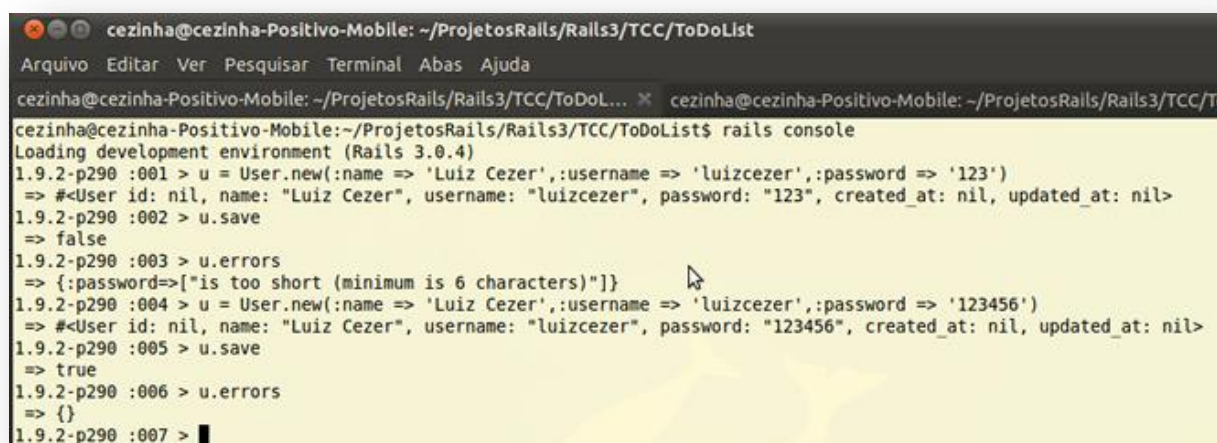
Fonte: Autoria Própria

A saída informa que não houve nenhum erro no fluxo testado, garantindo assim que a aplicação terá o comportamento esperando para todas suas funcionalidades.

4 RESULTADOS E DISCUSSÕES

Após a criação e verificação de todos os testes é preciso avaliar se o resultado final de aplicação foi o esperado, fazendo a verificação das validações via console e também por meio das telas do sistema.

Na interface, apresentada pela Figura 43, ao tentar criar um usuário com dados inválidos, às validações que antes foram testadas e criadas são chamadas garantindo que não haja um cadastro indevido de usuário e disparando as mensagens de erros adequadas. Caso o usuário seja validado nenhuma mensagem é exibida.

A screenshot of a terminal window titled 'cezinha@cezinha-Positivo-Mobile: ~/ProjetosRails/Rails3/TCC/ToDoList'. The terminal shows the execution of 'rails console' and the following commands and outputs:

```
cezinha@cezinha-Positivo-Mobile:~/ProjetosRails/Rails3/TCC/ToDoList$ rails console
Loading development environment (Rails 3.0.4)
1.9.2-p290 :001 > u = User.new(:name => 'Luiz Cezer', :username => 'luizcezer', :password => '123')
=> #<User id: nil, name: "Luiz Cezer", username: "luizcezer", password: "123", created_at: nil, updated_at: nil>
1.9.2-p290 :002 > u.save
=> false
1.9.2-p290 :003 > u.errors
=> {:password=>["is too short (minimum is 6 characters)"]}
1.9.2-p290 :004 > u = User.new(:name => 'Luiz Cezer', :username => 'luizcezer', :password => '123456')
=> #<User id: nil, name: "Luiz Cezer", username: "luizcezer", password: "123456", created_at: nil, updated_at: nil>
1.9.2-p290 :005 > u.save
=> true
1.9.2-p290 :006 > u.errors
=> {}
1.9.2-p290 :007 >
```

Figura 43 - Inserindo usuário via console

Fonte: Autoria Própria

Ao acessar a aplicação, a tela inicial mostra as tarefas já cadastradas (caso exista alguma), mas como o usuário não está autenticado somente a ação de *show* e um link para *login* são permitidos para o mesmo, conforme Figura 44.



Figura 44 - Acesso a aplicação sem autenticação

Fonte: Autoria Própria

Quando o usuário realiza sua autenticação na aplicação *links* para as ações de *edit*, *destroy* e *new* ficam visíveis para o usuário fazer o gerenciamento das tarefas e também um link para que seja efetuado o seu *logout*, conforme Figura 45.

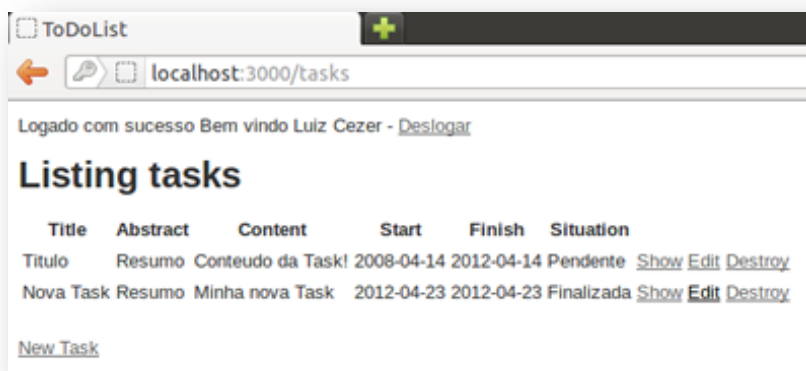


Figura 45 - Aplicação com usuário autenticado

Fonte: Autoria Própria

Para que a aplicação tivesse esse funcionamento foi preciso um pequeno ajuste na *view* de listagem de tarefas, visualizada na Figura 46.

Quando o valor de `session[:user_id]` não for nulo é por que há algum usuário logado na aplicação então os links para as ações *new*, *edit*, e *delete* ficam disponíveis.

```

<% @tasks.each do |task| %>
  <tr>
    <td><%= task.title %></td>
    <td><%= task.abstract %></td>
    <td><%= task.content %></td>
    <td><%= task.start %></td>
    <td><%= task.finish %></td>
    <td><%= task.situation %></td>
    <td><%= link_to 'Show', task %></td>
    <% unless session[:user_id].nil? %>
      <td><%= link_to 'Edit', edit_task_path(task) %></td>
      <td><%= link_to 'Destroy', task, :confirm => 'Are you sure?', :method => :delete %></td>
    <% end %>
  </tr>
<% end %>
</table>

<br />
<% unless session[:user_id].nil? %>
  <%= link_to 'New Task', new_task_path %>
<% end %>

```

Figura 46 – Ajuste de view de listagem de tarefas

Fonte: Autoria Própria

Na figura Figura 47 é apresentado um ajuste no template principal da aplicação, para que quando o usuário esteja autenticado, uma mensagem de boas vindas com seu nome seja exibida juntamente com um link de *logout*, caso contrário um link para que o usuário faça o *login*.

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>ToDoList</title>
5   <%= stylesheet_link_tag :all %>
6   <%= javascript_include_tag :defaults %>
7   <%= csrf_meta_tag %>
8 </head>
9 <body>
10 <%= flash[:notice] %>
11
12 <% if session[:user_id].nil? %>
13   <%= link_to 'Clique para logar', '/login' %>
14 <% else %>
15   Bem vindo <%= User.find(session[:user_id]).name %> - <%= link_to 'Deslogar', '/logout', :method => :delete %>
16 <% end %>
17 <%= yield %>
18
19 </body>
20 </html>
21

```

Figura 47 - Ajuste no template principal da aplicação

Fonte: Autoria Própria

Na linha 12 é feita a verificação para saber se há algum usuário autenticado, caso exista é exibida uma mensagem de boas vindas com um link de *logout*,

conforme a linha 15, caso não exista usuário logado é exibido um link para *login*, conforme a linha 13 que ao ser clicado redireciona o usuário para a página de *login*, que é visualizada na Figura 48.

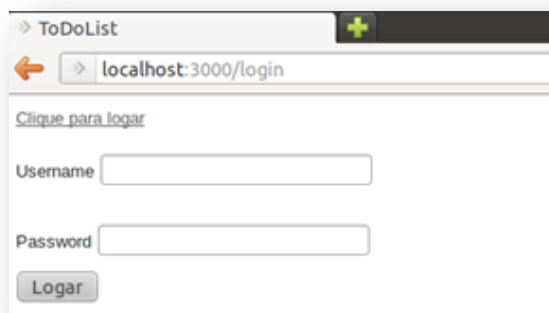


Figura 48 - Tela de login

Fonte: Autoria Própria

O usuário irá informar seus dados de username e password e tentará efetuar o *login* caso a autenticação seja realizada com sucesso, uma mensagem de boas vindas é exibida e o usuário é redirecionado para uma tela de criação de nova tarefa, conforme a Figura 49

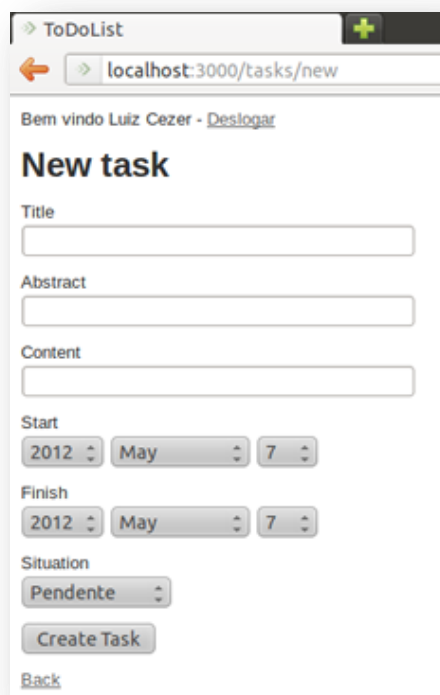


Figura 49 - Tela de adição de nova Task

Fonte: Autoria Própria

Com a autenticação bem sucedida o usuário é redirecionado para a página de cadastro de nova tarefa, ao topo é exibida uma mensagem de boas vindas com o nome de usuário logado e também um link para o seu *logout* e abaixado a tela onde os dados para cadastro da nova tarefa serão inseridos.

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÃO

Por meio de estudo do TDD foi possível conhecer como é a forma de desenvolvimento utilizando dessa técnica de teste e por meio da utilização da mesma juntamente com o *framework Rails* foi possível perceber a importância e os benefícios de se testar código antes de codificá-lo.

A linguagem Ruby juntamente com o *framework Rails* além de agilizarem a construção de aplicações web, também auxiliam o desenvolvimento de testes, pois a aplicação ao ser criada cria os diretórios que irão servir de ambiente de testes, onde o desenvolvedor poderá escrever tanto testes funcionais, unitários e de integração.

Escrevendo testes foi possível ter uma confiabilidade maior no código que viria a ser escrito, pois o mesmo além de funcional foi escrito de maneira limpa e sem código desnecessário o que além de deixá-lo mais legível poderá facilitar futuras manutenções ou melhorias.

À medida que mais testes são escritos o código é quebrado em mais partes, mantendo assim cada método simples e fazendo somente aquilo para qual foi criado.

Uso de TDD além de ajudar na codificação, ajuda a compreender melhor o sistema e seus comportamentos, pois ao escrever ou pensar nos testes são escritas pequenas histórias sobre o código de maneira que facilita o entendimento de qual é exatamente o comportamento que esperado de determinado método, facilitando dessa maneira o entendimento dos requisitos da aplicação.

5.2 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

A continuação desse trabalho sugere uma pesquisa mais elaborada sobre o uso de TDD em outras linguagens. Outra sugestão seria um estudo sobre como as linguagens ou frameworks facilitam a criação de testes no código.

6 REFERÊNCIAS BIBLIOGRÁFICAS

ANICHE, M. TDD realmente ajuda? **Pensamentos sobre desenvolvimento de software**, 16 Abril 2010. Disponível em: <<http://www.aniche.com.br/2010/04/tdd-realmente-ajuda/>>. Acesso em: 6 Fevereiro 2012.

BASTOS, L. Falando de TDD | Blog da Concrete. **Falando de TDD**, 17 Outubro 2012. Disponível em: <<http://blog.concretesolutions.com.br/2011/10/falando-de-tdd/>>. Acesso em: 7 Fevereiro 2012.

BECK, K. Manifesto para Desenvolvimento Ágil de Software. **Manifesto para Desenvolvimento Ágil de Software**, 2001. Disponível em: <<http://agilemanifesto.org/iso/ptbr/>>. Acesso em: 29 Fevereiro 2012.

BECK, K. **Test Driven Development By Example**. Boston: Addison-Wesley, 2002.

FOWLER, M. The New Methodology. **Martin Fowler**, Abril 2003. Disponível em: <<http://martinfowler.com/articles/newMethodology.html>>. Acesso em: 28 Fevereiro 2012.

FOWLER, M. The New Methodology. **Martin Fowler**, Abril 2003. Disponível em: <<http://martinfowler.com/articles/newMethodology.html>>. Acesso em: 28 Fevereiro 2012.

GUIA do Rails. **Guia do Rails - Um Guia para Testar Aplicações Rails**, 201-. Disponível em: <<http://guias.rubyonrails.com.br/testing.html>>. Acesso em: 13 Fevereiro 2012.

JANZEN, D. Software Architecture Improvement through Test-Driven Development., 2005.

KUHN, G. R. Apresentando XP. **Javafree**, 17 Agosto 2009. Disponível em: <<http://javafree.uol.com.br/artigo/871447/Apresentando-XP-Encante-seus-clientes-com-Extreme-Programming.html#introducao>>. Acesso em: 05 Março 2012.

LANGR, J. Evolution of Test and Code Via Test-First Design, 02 Dezembro 2002. Disponível em: <<http://www.objectmentor.com/resources/articles/tfd.pdf>>. Acesso em: 15 Março 2012.

LINGUAGEM.. **Linguagem de Programação Ruby**. Disponível em: <<http://www.ruby-lang.org/pt/>>. Acesso em: 19 Março 2012.

M., M. E.; WILLIAMS, L. Assessing test-driven development.

MAFRA, D. Introdução ao TDD. **Desenvolvimento com Qualidade e Efetividade**, 24 Setembro 2009. Disponível em: <<http://infoblogs.com.br/frame/goframe.action?contentId=192695>>. Acesso em: 13 Fevereiro 2012.

MARTINHO, F. TestExpert. **TestExpert**, 11 Março 2008. Disponível em: <<http://www.testexpert.com.br/?q=node/669>>. Acesso em: 10 Fevereiro 2012.

MONTEIRO, J. Ruby on Rails : Ruby on Rails Brasil. **Ruby on Rails Brasil**, 2012. Disponível em: <<http://rubyonrails.com.br/>>. Acesso em: 2012 Março 14.

PASSUELLO, L. A nova Metodologia - Martin Fowler. **Simplus**, Dezembro 2005. Disponível em: <<http://simplus.com.br/artigos/a-nova-metodologia/#N58>>. Acesso em: 28 Fevereiro 2012.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: MAKRON Books do Brasil Editora LTDA, 1995.

RIBEIRO, C. L. InfoQ : A relação entre TDD e Qualidade de Software. **InfoQ**, 5 Abril 2010. Disponível em: <<http://www.infoq.com/br/articles/relacao-tdd-qualidade>>. Acesso em: 6 Fevereiro 2012.

RIBEIRO, C. L. InfoQ : A relação entre TDD e Qualidade de Software. **InfoQ**, 5 Abril 2010. Disponível em: <<http://www.infoq.com/br/articles/relacao-tdd-qualidade>>. Acesso em: 6 Fevereiro 2012.

SANCHEZ, I. Introdução do Desenvolvimento voltado a Testes (TDD) < Coding Dojo Floripa. **Coding Dojo Floripa**, 7 Novembro 2006. Disponível em: <<http://dojofloripa.wordpress.com/2006/11/07/introducao-ao-desenvolvimento-orientado-a-testes/>>. Acesso em: 2 Fevereiro 2012.

SANTOS, W. NetFeijão Brasil TDD: Test Driven Development. **NetFeijão**, 3 Janeiro 2008. Disponível em: <<http://netfeijao.blogspot.com/2008/01/tdd-test-driven-development.html>>. Acesso em: 13 Fevereiro 2012.

SEA, T. Minicurso de TestesOnRails. **Slideshare**, 10 Agosto 2009. Disponível em: <<http://www.slideshare.net/seatecnologia/minicurso-de-testesonrails>>. Acesso em: 14 Fevereiro 2012.

TELES, V. M. Extreme Programming, XP: metodologia de desenvolvimento ágil. **Improve It**, 2008. Disponível em: <<http://improveit.com.br/xp/>>. Acesso em: 05 Março 2012.

THOMAS, D. H. **Desenvolvimento Web Ágil com Rails**. Porto Alegre: Bookman, 2008.