



**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO
TOCANTINS
CAMPUS ARAGUATINS
CURSO SUPERIOR DE PÓS-GRADUAÇÃO LATO SENSU EM
DESENVOLVIMENTO DE SISTEMAS
COMPUTACIONAIS**

LEANDRO GOMES CARDOSO

**PRÁTICAS DE DESENVOLVIMENTO DE SISTEMAS UTILIZANDO TESTES
AUTOMATIZADOS PARA MELHORIA DA
QUALIDADE DE *SOFTWARE***

Araguatins-TO
2021

LEANDRO GOMES CARDOSO

**PRÁTICAS DE DESENVOLVIMENTO DE SISTEMAS UTILIZANDO TESTES
AUTOMATIZADOS PARA MELHORIA DA
QUALIDADE DE *SOFTWARE***

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de pós-graduação lato sensu em desenvolvimento de sistemas computacionais da Unidade *campus* Araguatins, do Instituto Federal do Tocantins, como exigência à obtenção do título de Especialista em Desenvolvimento de Sistemas Computacionais.

Orientador: Me. Ancelmo Frank Coelho Castro

Araguatins-TO
2021

Dados Internacionais de Catalogação na Publicação (CIP)
Bibliotecas do Instituto Federal do Tocantins

- C268p Cardoso, Leandro Gomes
 Práticas de desenvolvimento de sistemas utilizando testes
 automatizados para melhoria da Qualidade de Software / Leandro
 Gomes Cardoso. – Araguatins, TO, 2021.
 26 f. : il. color.
- Artigo (Especialização em Desenvolvimento de Sistemas
 Computacionais) – Instituto Federal de Educação, Ciência e
 Tecnologia do Tocantins, Campus Araguatins, Araguatins, TO, 2021.
- Orientador: Me. Ancelmo Frank Coelho Castro
1. TDD. 2. Desenvolvimento. 3. Elixir. I. Castro, Ancelmo Frank
 Coelho. II. Título.

CDD 004

A reprodução total ou parcial, de qualquer forma ou por qualquer meio, deste documento é autorizada para fins de estudo e pesquisa, desde que citada a fonte.

Elaborado pelo sistema de geração automática de ficha catalográfica do IFTO com os dados fornecidos pelo(a) autor(a).

LEANDRO GOMES CARDOSO

**PRÁTICAS DE DESENVOLVIMENTO DE SISTEMAS UTILIZANDO TESTES
AUTOMATIZADOS PARA MELHORIA DA
QUALIDADE DE SOFTWARE**

Trabalho de Conclusão de Curso apresentado à
Coordenação do Curso de pós-graduação lato
sensu em desenvolvimento de sistemas
computacionais da Unidade *campus* Araguatins,
do Instituto Federal do Tocantins, como exigência
à obtenção do título de Especialista em
Desenvolvimento de Sistemas Computacionais.

Aprovado em: 08/04/2021

BANCA EXAMINADORA

Me. Ancelmo Frank Coelho Castro
Orientador

*Instituto Federal De Educação, Ciência E Tecnologia Do Tocantins
Campus Araguatins*

Me. Adeilson Marques da Silva Cardoso
Instituto Federal De Educação, Ciência E Tecnologia Do Tocantins
Campus Araguatins

Me. Marco Aurelio Gomes de Oliveira
Instituto Federal De Educação, Ciência E Tecnologia Do Tocantins
Campus Araguatins

AGRADECIMENTOS

Agradeço a Deus primeiramente por ter me dado saúde e inteligência para a trilha e passar por todas as dificuldades e conseguir esta conquista.

A minha família agradece por estar sempre comigo participando de cada momento de minha trilha, mas em especial agradeço aos meus pais, Elizete da Silva Gomes e Juarez Dias Cardoso pela grande força e conselho como também sacrifícios para alcançar meus sonhos.

Ao meu professor orientador Me. Ancelmo Frank Coelho Castro por ter me ajudado e colaborado para tornar a elaboração deste trabalho possível, com suas orientações, paciência e dedicação.

Ao Instituto Federal de Educação, Ciência e Tecnologia do Tocantins – Campus Araguatins por oferecer um ambiente excelente aos alunos, tornando realidade um sonho.

Agradeço aos colegas de curso por contribuir com os momentos que tivemos juntos.

RESUMO

O processo de desenvolvimento de *software* evoluiu muito com o passar do tempo, tendo em vista o máximo de qualidade, a forma como se desenvolvia *software* ficou ultrapassada como o tempo sendo necessário adaptações para melhorias, uma destas adaptação e a utilização de testes automatizados no processo de desenvolvimento de *software* como forma de garantir a qualidade e constante melhorias no código. Como esta ideia, este trabalho vem como um propósito de apresentar o processo de desenvolvimento de um sistema com a utilização de testes automatizados por meio da prática de TDD(*Test-driven Development* ou Desenvolvimento Orientado a Testes), que é uma forma de utilização de testes automatizados que prega que os testes têm que ser escritos antes do código a ser testado. No final do processo de desenvolvimento serão analisados os benefícios que o TDD traz ao processo de desenvolvimento.

Palavras-chave: TDD. Testes automatizados. Desenvolvimento. Elixir.

ABSTRACT

The software development process has evolved a lot over time, in view of the maximum quality, the way in which software was developed has been outdated as time requires adaptations for improvements, one of these adaptations and the use of automated tests in the process software development as a way to ensure quality and constant code improvements. Like this idea, this work comes as a purpose to present the process of developing a system with the use of automated tests through the practice of TDD (Test-driven Development), which is a way of using automated tests that state that tests have to be written before the code to be tested. At the end of the development process, the benefits that TDD brings to the development process will be analyzed.

Keywords: TDD. Automated testing. Development. Elixir.

LISTA DE ILUSTRAÇÕES

Figura 1	– Ciclo do TDD	13
Figura 2	– Modelo das entidades	15
Figura 3	– Geração da estrutura do projeto	16
Figura 4	– Adição das dependências extras	17
Figura 5	– Comando para gerar arquivos base do user	18
Figura 6	– Configurações iniciais para o testes	19
Figura 7	– Teste de verificação do atributo active	20
Figura 8	– Resultado do teste no terminal	20
Figura 9	– Resultado do teste de criar novo agendamento	22
Figura 10	– função create_event	23
Figura 11	– função create_event refatorado	24

LISTA DE SIGLAS

API	Application Programming Interface ou Interface de Programação de Aplicativos
CLI	<i>Command Line Interface</i> ou <i>Interface de Linha de Comando</i>
MVC	<i>Model View Controller</i> ou <i>Modelo Visualização Controle</i>
TDD	<i>Test-driven Development</i> ou Desenvolvimento Orientado a Testes

SUMÁRIO

1 INTRODUÇÃO	11
2 REFERENCIAL TEÓRICO	11
2.1 O porquê de testar software?	11
2.2 Testes automatizados	12
2.3 Test-driven Development (TDD)	12
3 DEFINIÇÃO DAS TECNOLOGIAS	14
4 DESENVOLVIMENTO	14
4.1 Criação e configuração inicial do projeto	16
4.2 Codificação da Web API seguindo o TDD	17
4.3 O primeiro módulo de teste	18
4.4 Testes complexos do software	21
5 CONSIDERAÇÕES FINAIS	24
REFERÊNCIAS	26

1 INTRODUÇÃO

A forma com a qual desenvolvemos sistemas está em constante evolução, passando de sistemas sem qualquer forma de testes, para sistemas onde até mesmo formulários de páginas web tem uma forma de interação automatizada, sendo preenchidas e submetidas. Segundo Aniche (2012, p. 3) para se ter um sistema totalmente testado de forma constante e contínua é utilizado testes automatizados.

Tendo em mente que testar *software* muitas vezes não é uma tarefa fácil, principalmente quando se faz manualmente, pois necessita de muita atenção e repetição de ações para sanar essa dificuldade, temos os testes automatizados, que possuem várias práticas de codificação.

Uma delas é a utilização de TDD (*Test-driven Development* ou Desenvolvimento Orientado a Testes) que será abordada neste artigo.

Este trabalho apresenta uma proposta prática da implementação do TDD no desenvolvimento de uma *Web API*, com objetivo de demonstrar o processo de criação e utilização de testes automatizados para códigos específicos dentro de um determinado modelo de um protótipo de uma *Web API* por meio de testes unitário, como também mostrar os benefícios ao utilizar testes automatizados para se ter uma melhoria contínua do *software* com garantia de que ele não irá sofrer com uma falha causada pelas melhorias implementadas, já que os teste garante que após a modificação todos código continua executando em perfeita condição.

Para isso será apresentado desde o início da prototipação a utilização de testes automatizados, a fim de ter um software testado logo no seu começo.

Neste artigo temos as seguintes seções: fundamentação sobre o TDD e tecnologias usadas no desenvolvimento, processo de desenvolvimento guiado por testes da API e por fim as considerações finais e trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 O porquê de testar *software*?

A resposta é simples. Ao entregar um *software* sem testes existe uma grande probabilidade de algo dar errado na hora da sua execução. Exemplo: um sistema bancário onde grande quantidade de dinheiro trafega de uma conta para outra. Imagine se um dos métodos executa de forma indevida gerando dois depósitos ao invés de um só?.

Um exemplo real deste tipo de erro pode ser encontrado em matérias de revista e/ou sites como a seguinte: “Itaú comete falha com o Pix, transfere R\$1 milhão indevidamente e processa bancos”(GUSSON, 2021). A matéria também cita que o valor das transferências indevidas totalizam exatos R\$ 966.392,00 reais.

Como isso podemos ver a importância de testes automatizados em *softwares*, que mesmo com suas dificuldades em introdução no início de um projeto e/ou para uma equipe inexperiente com teste, trazem grandes benefícios.

2.2 Testes automatizados

A forma mais recomendada de se testar *software* é utilizado testes automatizados que são *script* ou código na própria linguagem de programação que automatizam o processo no teste de *software* garantido sempre que o mesmo mantenha seu padrão e qualidade pois antes de se entregar o código em produção e feito um teste em todas as classes assim assegurado que a alteração feita não prejudicou o restante do sistema.

Segundo Lima(2019 p.39) o conceito de testes automatizados e rotinas que têm pouca ou nenhuma interação humana mas simula várias funcionalidades presentes no *software*, validando todas as respostas e possíveis erros.

Há várias formas de se testar um código ou conjunto de código, já que dependendo do resultado a ser obtido é necessária a utilização de teste de uma maneira diferente são exemplos de tipos de testes os a seguir:

- Teste Unitário: responsável por testar pequenas partes do sistema(unidade) como uma função ou módulo(classe)
- Teste de Integração: testa a junção de dois o mais módulos(classes) que juntos resolvem um determinado problema, por meio de excursão da função do ponto inicial ao ponto final garantindo que mesmo que passando por todos módulos(classes) o resultado é o esperado

2.3 Test-driven Development (TDD)

O desenvolvimento de *software* em muito adaptativo e no testes automatizados não são diferente o mesmo garante sua utilização de várias maneiras no processo de desenvolvimento como por exemplo testa somente no final do sistema já pronto, testa ao fim de cada funcionalidade ou iniciar a programação pelo testes essa prática é conhecida como TDD.

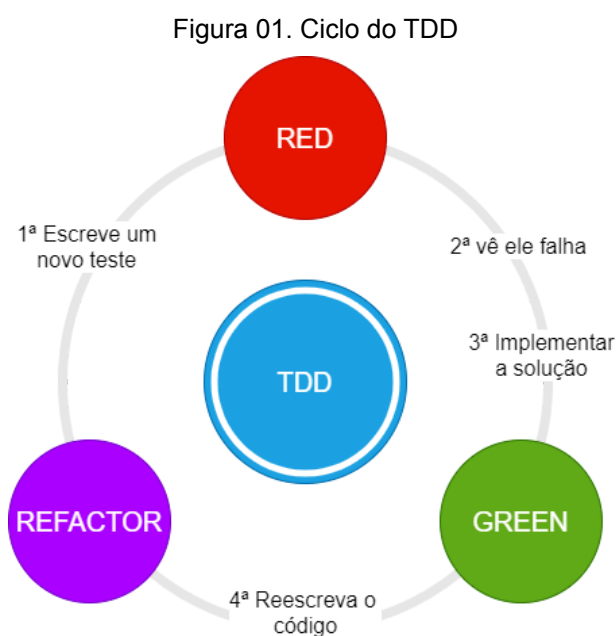
O TDD surgiu de uma necessidade de fazer testes em aplicações desenvolvidas como a utilização de métodos ágeis onde tem uma entrega contínua do *software* por esse motivo não se podia fazer o teste somente no final do desenvolvimento. A partir da década 1990 como surgimento dos métodos ágeis, como não havia sentido em se testar no final, pois as entregas eram constantes buscou-se novas formas de testar software.(BARROS E JUNIOR, 2019, p.23).

Guedes (2019) argumenta que o TDD ou Desenvolvimento Orientado a Testes, tem como principais objetivos garantir que o software funcione corretamente e com qualidade e entrega no prazo correto, trabalhando em forma de ciclo seguindo uma ordem específica sendo: primeiro escreve o teste, segundo fazer o código

passar e terceiro refatorar o código. O TDD torna mais fácil a vida do desenvolvedor, pois já que ele se torna uma parte no processo de desenvolvimento e não uma parte extra que é executada após o término do desenvolvimento.

Quando se tem uma tradução direta de *Test-driven Development* temos Desenvolvimento Orientado a Testes, ou seja como o próprio nome já informa os testes são implementados antes mesmo do código de produção. A ideia é que o desenvolvedor inicie a implementação pelo teste visando sempre código simples e de qualidade (ANICHE 2012, p. 25).

O TDD tem um ciclo bem definido conhecido como red-green-refactor, se inicia no red que o teste que falha passa para o green que é quando o código passa mas sendo uma versão simples da implementação por último temos o refactor para a melhorias no código.



Fonte: O autor

- Escreva um novo teste: o ciclo se inicia pelo teste até mesmo antes de implementar o código a ser testado.
- Vê o teste falha: como ainda não se tem o código que vai ser testado vai gerar um erro e com base no mesmo pode se tirar uma série de informações nas quais podem influenciar na implementação ou não do código.
- Implementar a solução: após ver o erro que foi gerado se tem uma ideia que codifica uma resolução assim é gerado o código mais simples possível para passar no teste.
- Reescreva o código: a primeira versão da solução é um código simples feito para passar então pode ser feito melhorias.

3 DEFINIÇÃO DAS TECNOLOGIAS

Para desenvolver o software de maneira mais simplificada é necessário a escolha de tecnologias que tragam mais facilidade de se programar utilizando as práticas de TDD, além de uma melhor compreensão do código escrito.

Sendo assim, para este artigo, foi escolhida a linguagem de programação Elixir e o *framework* Phoenix. Ambos possuem embutido o sistema completo de testes, não sendo necessária a inclusão de novas dependências, a não ser para melhorar alguma forma de teste.

Elixir é uma linguagem de programação funcional, moderna e de alto desempenho. Foi criada pelo brasileiro José Valim. Roda em cima da máquina virtual do Erlang (*linguagem de programação criada pela Ericsson, destinada a aplicações de grande porte, por exemplo WhatsApp*) que tem como propósitos: escalabilidade, tolerância a falhas e alta concorrência. Ela aumenta a capacidade do Erlang, trazendo novas funcionalidades e reduzindo sua complexidade, uma vez que ele traz uma sintaxe um pouco difícil de ser compreendida, diminuindo assim a curva de aprendizagem (DAVI 2017, p. 1).

Phoenix é um *framework* completo, *open source*, feito para a Elixir; segue o padrão MVC (*Model View Controller ou Modelo-Visualização-Controle*), com todos os padrões modernos implantados, abstraindo códigos, facilitando a programação e que pode ser utilizado na criação de *API's*, aplicações web entre outros.

O paradigma de programação funcional, diferente de outras linguagens, que trazem a orientação a objetos como seu paradigma, é adotado na Elixir. Ela é 100% funcional, empregando todos os conceitos, desde a sua criação. Segundo Davi (2017 p. 3), programar funcionalmente é a transformação dos dados através de funções, sendo parte principal dentro do código, composta por funções menores. A Elixir possui um *framework* de testes, completo, integrado: o ExUnit. Os testes são feitos em arquivos do tipo *script* com extensão (.exs), como o phoenix, que tem total integração com a suíte de testes padrão da Elixir.

4 DESENVOLVIMENTO

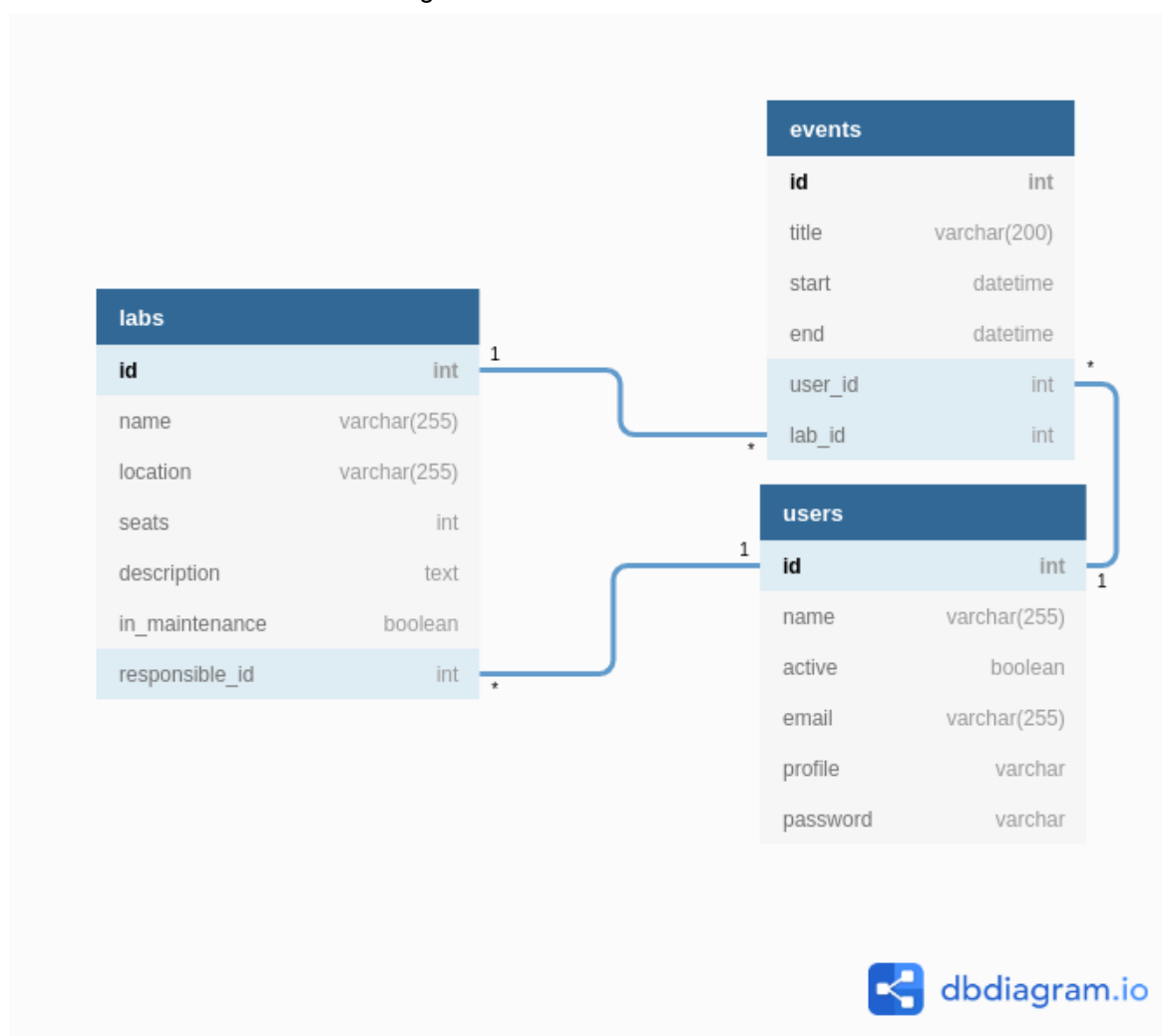
Para se ter um sistema bem definido, existem os importantes passos a serem seguidos. Após o levantamento de requisitos é importante que se faça a modelagem do *software*, garantindo assim que não sejam feitos trabalhos desnecessários, quando os requisitos não estão bem definidos.

O sistema proposto como exemplo, para desenvolvimento deste artigo, foi o de controle de agendamento de laboratórios dentro de uma instituição de ensino,

visando facilitar o acesso às informações sobre um determinado laboratório, assim podendo ser agendado em um dia que esteja disponível.

Seguindo essa visão, os primeiros passos para construção da *Web API* são: a modelagem do sistema, definição do banco de dados, entidades e relacionamentos. Para facilitar a modelagem, foi usado o *software* DBDiagram, disponível no endereço (<https://dbdiagram.io/home>). O resultado pode ser visto na figura 02.

Figura 02. Modelo das entidades



Fonte: O autor

Alguns padrões de modelagem podem ser vistos nesta figura 02, com a utilização dos nomes das tabelas no plural e em inglês, bem como suas colunas, para garantir que não ocorra erro ao tentar criar as tabelas, uma vez que os bancos não trabalham bem com acentuação e caracteres especiais.

4.1 Criação e configuração inicial do projeto

A Criação do projeto é feito através da CLI (*Command Line Interface* ou *Interface de Linha de Comando*), instalada no sistema operacional e acessada via terminal (bash, cmd, powershell). Ela disponibiliza vários comandos que ajudam na construção de *software*. A seguir, na figura 03, mostra como é feita a geração do projeto.

Figura 03. Geração da estrutura do projeto



Fonte: O autor

O comando acima leva a alguns *flags*. O primeiro indica que o projeto não utilizará o *webpack* (empacotador de código do front-end) e a segunda, que não vai usar HTML, já que o mesmo se trata de uma *Web API*. Após isso, temos como resultado uma estrutura de diretórios como a demonstrada abaixo:

— _build	-- Pasta com os arquivos de produção
— config	-- Pasta com as configurações do <i>software</i>
— deps	-- Pasta com as dependências do <i>software</i>
— lib	-- Pasta destinada aos códigos de desenvolvimento
— mix.exs	-- Arquivo de gerenciamento do projeto
— mix.lock	-- Grava a versões das dependências de quando foi gerado
— priv	-- Pasta para arquivos de produção como o de tradução
— README.md	-- Descrição do projeto para controle de versão
— test	-- Pasta que conterà todos os <i>scripts</i> e configurações de testes. Essa pasta, assim como a <i>lib</i> , são as mais importantes para este artigo.

Já pensando em manter o projeto com código de qualidade, serão adicionadas duas dependências:

- Credo: analisador de escrita e consistência de código
- ExCoveralls: gerador de relatórios de cobertura de testes

Com essas duas dependências temos um total controle sobre a escrita de código, deixando-o, desde o início, com excelente qualidade e, também, sabendo onde ainda é necessário fazer testes para aumentar a cobertura.

O processo de adição destas duas dependências é simples: basta adicionar linhas referente à elas na função responsável, dentro do arquivo `mix.exs`, responsável tanto pelo gerenciamento das dependências do projeto como também por algumas configurações.

Figura 04. Adição das dependências extras



Fonte: O autor

A figura 04 demonstra onde se adiciona as dependências do projeto. Nela podemos ver informações importantes como versão e seu ambiente de destino. Após isso, é necessário apenas executar no terminal o comando ***mix deps.get***.

A última configuração antes do início da codificação é a de acesso ao banco de dados. Os arquivos responsáveis pelas configurações de banco de dados, dentro da pasta ***config***, são: *dev.exs* e *test.exs*, que representam cada um ambiente contendo todas as configurações necessárias para o seu funcionamento. Estes arquivos foram feitas as alterações do *username* (*usuário*) e *password* (*senha*) para o acesso ao banco de dados, já que ele vem por padrão com o banco dados PostgreSQL.

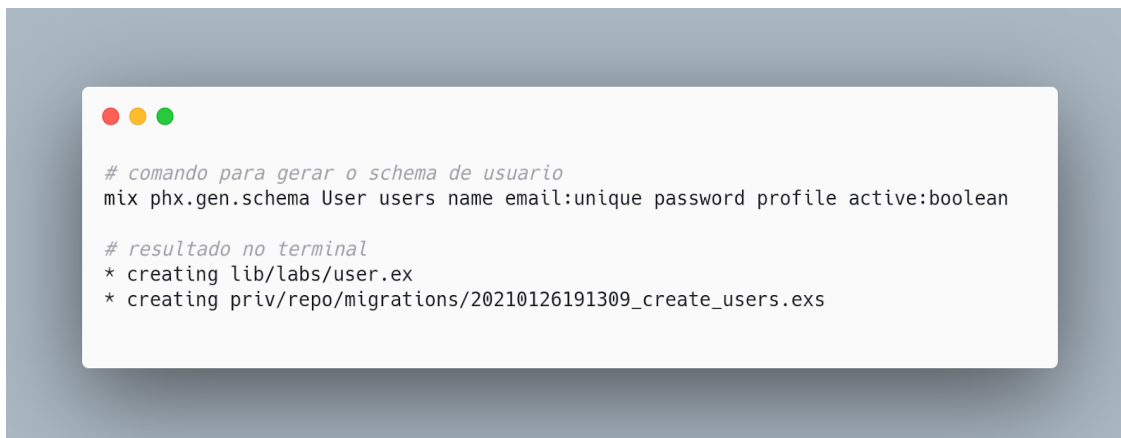
4.2 Codificação da Web API seguindo o TDD

Antes da criação do primeiro teste tem algumas informações importantes, sobre o processo de desenvolvimento deste protótipo, visando ser um trabalho mais simplificado aqui será utilizado somente testes unitários, por serem mais simples de se criar e explicar.

Analisando o modelo de banco de dados visto na figura 02, podemos definir que os testes e implementação vão começar pelo modelo de user (Usuário), já que é o único que não possui chave estrangeira (Foreign Key/FK). Então, para isso, é

possível apenas rodar um comando que gera os arquivos necessários como visto na figura 05.

Figura 05. Comando para gerar arquivos base do user



```
# comando para gerar o schema de usuario
mix phx.gen.schema User users name email:unique password profile active:boolean

# resultado no terminal
* creating lib/labs/user.ex
* creating priv/repo/migrations/20210126191309_create_users.exs
```

Fonte: O autor

A primeira linha da figura 05 é o comando de geração de *schema* onde recebe como parâmetros nome do *schema*, *nome da tabela* na base de dados e *atributos*. Alguns deles têm marcação diferente, como o *email*, e recebe o *unique*, mostrando que ele é um campo único na tabela, e o *active*, que muda seu tipo, que por padrão é *String*, para o *Boolean*.

O resultado no terminal mostra que foram criados dois arquivos: o arquivo *user.ex*, que é o *schema* propriamente dito, e o arquivo de *migration*, que tem a extensão *.exs*. Esta não só é a representação do banco de dados com todas as colunas e seus tipos, como também configurações de coluna única e valor padrão para a coluna do tipo Boolean.

4.3 O primeiro módulo de teste

Elixir, por ser uma linguagem funcional, não possui o conceito de classes. Em seu lugar, as agrupações de funções ficam dentro do que é chamado de módulo, que tem outras diferenças de classes em outras linguagens de programação, mas que não serão abordadas aqui.

Antes de codificar o método de teste algumas configurações para facilitar o teste, estas podem ser vistas a seguir na figura 06. Os testes na Elixir serão descritos aqui em cada função das linhas da figura a seguir.

- Linha 1: Definição do módulo de teste
- Linha 2: Requisição da ajuda de testes relacionado a dados
- Linhas 4 e 5: Criar um encurtado “*User*” e para o “*Repo*”

- Linha 6: Descreve o início de um bloco de teste
- As linhas 8 a 10: criar variáveis para uso dentro do bloco os dados de dentro foram omitidas para melhor entendimento
- Linha 12: Definição de um teste onde ir conter o código de teste, por padrão o nome dado ao teste tem que ser referente ao que ele faz
- Linha 13: Criado um changeset que como um construto de um *schema* do tipo User devendo assim um *schema* válido
- As linhas 14 e 15: são feitas assertivas, comparando os valores desejados como os obtidos.

Figura 06. Configurações iniciais para o testes



```

1 defmodule Labs.UserTest do
2   use Labs.DataCase
3
4   alias Labs.User
5   alias Labs.Repo
6
7   describe "users" do
8     @valid_attrs %{...}
9     @update_attrs %{...}
10    @invalid_attrs %{...}
11
12    test "insert/1 with valid data creates a user" do
13      changeset = User.changeset(%User{}, @valid_attrs)
14      assert {:ok, %User{} = user} = Repo.insert(changeset)
15      assert user.name == "leandro"
16    end
17  end
18 end

```

Fonte: O autor

Este teste passa mesmo sem qualquer alteração na implementação. Isto se dá por que, ao gerar o módulo *User* pelo terminal, ele já vem com vários códigos prontos e validações, sendo necessárias somente as adaptações aos casos específicos de regra de negócio. Por estes testes já passarem eles serão omitidos partindo para os testes que falham na execução.

O próximo teste será verificar se, ao salvar o novo usuário sem passar o atributo *active*, ele salva por padrão como *True*, já que na regra de negócios foi definido que o usuário já vai ser criado ativo.

As figuras 07 e 08 mostram respectivamente o código de teste e o seu resultado no terminal.

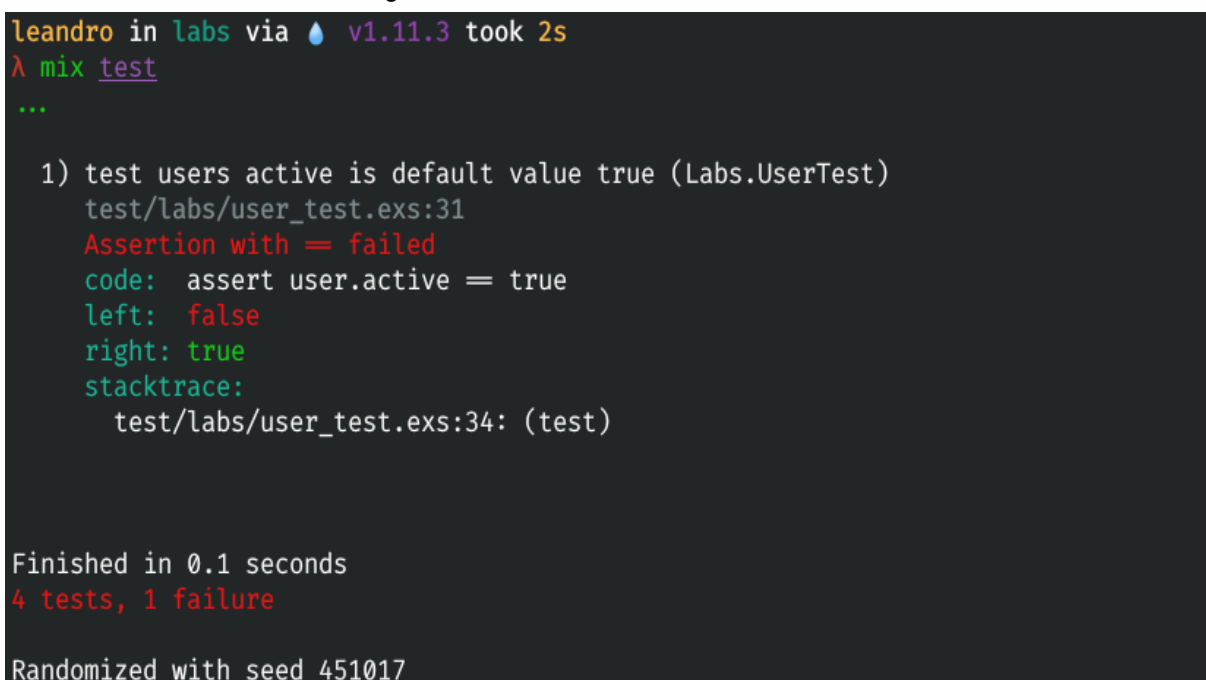
Figura 07. Teste de verificação do atributo active



Fonte: O autor

A figura 07 mostra todos os passos do teste de verificação do atributo *active*. As linhas mais importantes são a 4 e 6. A primeira cria um *changeset*, passando os atributos válidos somente removendo o *active* que é feito pelo *Map.delete*. Em sequência ele é inserido no banco. Na linha 6 é feita a comparação do atributo *active* do usuário retornado com o valor *true*.

Figura 08. Resultado do teste no terminal



Fonte: O autor

A figura 08 traz o resultado da execução no terminal do comando *mix test*. Ele traz descrito todos os testes que falham. Podemos ver que o único a falhar é o que está codificado na figura 06. As informações que se encontram na figura 8, são

muito importantes para o entendimento do porque falhou o teste. Podemos encontrar o nome do teste, *code* que indica a linha de código que não passou no teste e logo abaixo temos: *left* valor resultante e o *right* valor esperado.

Para fazer esse teste passar é necessário uma pequena modificação no módulo de *user*, alterando no atributo *active* o valor de *default*, de *False* para *True*.

Após essa introdução sobre como fazer testes na Elixir e Phoenix, podemos partir para os testes mais complexos presentes na regra de negócio, já que o foco principal deste sistema é fazer agendamentos de laboratórios para datas disponíveis.

4.4 Testes complexos do software

Pelo fato do agendamento ser a parte mais complexa no desenvolvimento do sistema, para seguir as boas práticas de programação, este módulo será gerado de uma forma diferente. Ao invés de gerar apenas o *Schema* utilizado, com comando *mix phx.gen.schema*, será usado o comando *mix phx.gen.context*. O porquê de utilizar *Context* (*Contexto*) no lugar de *Schema*? O motivo é a abstração de todo código complexo que iria para o controle em um módulo diferente. Esse é o *Context*.

E quais cenários de testes que podem ser analisados? Teremos alguns cenários. Dentre eles, criações de agendamentos com a data em uso, agendamentos com a data em uso em outro laboratório, listagem de todos os agendamentos de um determinado laboratório, entre outros.

Para exemplificação de um teste complexo, será demonstrado abaixo o processo de desenvolvimento seguindo o TDD do cenário de criação de um novo agendamento levando em consideração todas as regras de negócio, como: o não agendamento de um evento em uma mesma data-hora e local de uma evento já cadastrado. Mas ao realizar o teste vemos que o valor esperado não corresponde com o valor resultante como demonstra a figura 09.

Figura 09. Resultado do teste de criar novo agendamento

```

1) test events create_event/1 with valid data and usage data returns error (Labs.SchedulesTest)
   test/labs/schedules_test.exs:59
   match (=) failed
   code: assert {:error, _} = Schedules.create_event(@valid_attrs, lab)
   left:  {:error, _}
   right: {
           :ok,
           %Labs.Schedules.Event{__meta__: #Ecto.Schema.Metadata<:loaded, "events">, end: ~N[2010-01-28 17:27:41], lab: %Labs.Lab{__meta__: #Ecto.Schema.Metadata<:loaded, "labs">, description: "local", name: "Lab Programação", responsible_id: nil, seats: "10", updated_at: ~N[2021-01-28 17:27:41], user: %Labs.User{__meta__: #Ecto.Schema.Metadata<:loaded, "users">, active: true, email: "user@example.com", name: "User", password: "password", profile: nil, updated_at: ~N[2021-01-28 17:27:41]}, use
   stacktrace:
     test/labs/schedules_test.exs:65: (test)

```

Fonte: O autor

Ao analisar o resultado do teste, temos a informação de que se espera uma *tupla* (um tipo de dado encontrado no Elixir e se compara com uma lista) que retorna um *atom* (uma constante da Elixir) de `:error` e um valor qualquer que foi omitido como indicado na linha do *left* da figura, mas o que é retornado da chamada da função `create_event` e um símbolo `:ok` mais um *event* que acabou de ser salvo em banco de dados. Mas este não é o comportamento esperado, isso ocorre porque essa regra de não salvar um evento na mesma data-hora e local é específica deste projeto, assim sendo necessário a alteração de seu comportamento padrão para que atenda às necessidades. Essa alteração pode ser vista na figura 10.

Figura 10. função create_event

```

1 def create_event(attrs, lab, user) do
2   events = Repo.all(
3     from e in Event,
4     where: e.lab_id == ^id and e.start <= ^attrs.start
5   )
6
7   available = case Enum.empty?(events) do
8     true ->
9       true
10
11     false ->
12       for event <- events do
13         if event.start >= attrs.start or event.end >= attrs.end do
14           false
15         else
16           true
17         end
18       end
19   end
20
21   if available == true do
22     %Event{}
23     |> Event.changeset(attrs)
24     |> Changeset.put_assoc(:lab, lab)
25     |> Changeset.put_assoc(:user, user)
26     |> Repo.insert()
27   else
28     {:error, "data hora já em uso para este laboratorio"}
29   end
30 end
31 end

```

Fonte: O autor

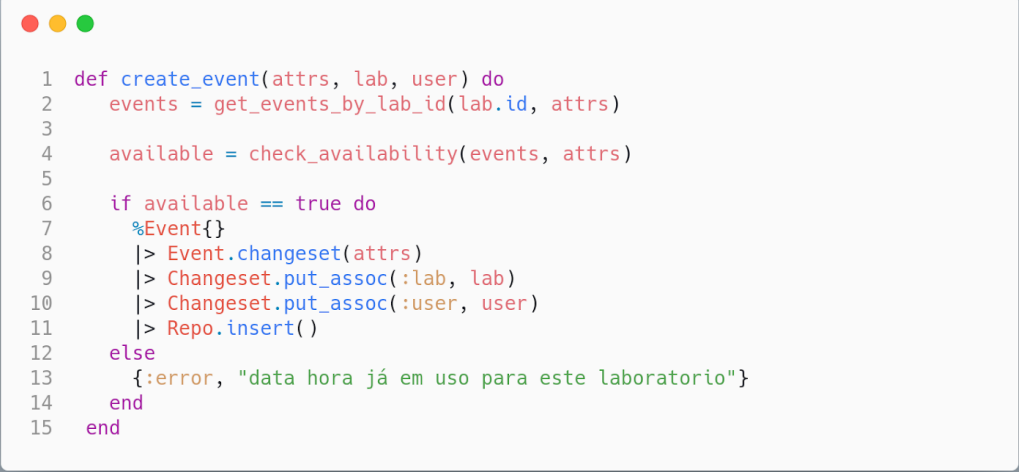
Percebe-se que agora está sendo feita a verificação de que se há um evento cadastrado para a data-hora e local selecionado, esse código pode ser visto entre as linhas 2 a 19, devolvendo assim um *true*, se está disponível, e um *false*, se não está. Com esse resultado, é feito um *if (se) else (senão)* das linhas 22 a 30, onde o *if* verifica se o valor contido dentro de *available*(linha 7) e *true* se for passa pelo processo de criação senão retorna *tupla* contendo um *atom (:error)* e uma mensagem(linha 29).

Mas como foi visto anteriormente, o TDD tem um ciclo definido de ações onde, neste exemplo de teste, já foi passado por duas das três fases, sendo a próxima a refatoração. Mas o que pode ser melhorado do código da figura 10? Se analisarmos com cuidado vemos que ele quebra um dos princípios da qualidade de código: o de que uma função deve ter apenas uma responsabilidade, porém vemos que não é o caso, porque ela não só cria o evento como também faz as verificações.

Como melhorar então o código? De forma simples, é só fazer a abstração do código de verificação, assim garantido que cada função será responsável por

apenas uma responsabilidade. Uma das facilidades obtidas quando se faz TDD é que a alteração do código é garantida com os testes, ou seja, mesmo com alterações do código, temos uma garantia de que não haverá um erro implementado. Sendo assim, pode-se ver o resultado da refatoração na figura 11.

Figura 11. função create_event refatorado



```
1 def create_event(attrs, lab, user) do
2   events = get_events_by_lab_id(lab.id, attrs)
3
4   available = check_availability(events, attrs)
5
6   if available == true do
7     %Event{}
8     |> Event.changeset(attrs)
9     |> Changeset.put_assoc(:lab, lab)
10    |> Changeset.put_assoc(:user, user)
11    |> Repo.insert()
12   else
13     {:error, "data hora já em uso para este laboratorio"}
14   end
15 end
```

Fonte: O autor

O que pode ser observado na figura 11 é que o código da função ficou de um tamanho menor e com uma só responsabilidade, fazendo chamada de outras funções quando necessário. (linhas 2 e 4). Ao realizar o teste desta função novamente, vemos que a mesma passa com sucesso isso se dar por que a alteração realizada não trouxe qualquer mudança que quebra a execução do código.

5 CONSIDERAÇÕES FINAIS

O processo de criação de testes automatizados utilizando a linguagem de programação Elixir e o Phoenix *framework* é em sua grande parte simples, já que trazem facilitadores que ajudam no processo de codificação pois traz configurações por padrão para utilização imediata. Em quando a execução dos testes é rápida somente necessário execução de uma CLI e traz várias informações em caso de erro e em caso de sucesso mostra a lista de testes executada com sucesso.

Após do teste criado tem se a garantia de que o código tem que agir de uma determinada forma para não gerar erro, como isso permitindo melhorias contínuas no software, além de garantir que não ocorra erros não tratados para o usuário.

Então, conforme exposto durante este trabalho, testar um *software* de forma automatizada traz grandes benefícios: qualidade de código, garantia de refatoração com segurança, ainda mais quando as tecnologias empregadas no desenvolvimento trazem total suporte aos testes automatizados como as aqui apresentadas.

Isso porque os testes automatizados trazem segurança de que se uma melhoria for mal implementada, ela vai resultar em um erro na execução dos testes, assim obrigando a verificação das mudanças e a correção do mesmo.

Todos os objetivos propostos foram alcançados de forma simplificada. O principal motivo desta simplicidade foi a escolha das tecnologias empregadas no desenvolvimento, uma vez que a mesma garante a construção de *software* completamente testados.

REFERÊNCIAS

- ANICHE, Mauricio. **Test-Driven Development: Teste e Design no Mundo Real**. 1. ed. Vila Mariana – São Paulo – SP – Brasil: Casa do Código, 2012. 194 p. *E-book*.
- BARROS, Gabryela Santana; JUNIOR, Lielson R. Pereira. **Análise experimental entre as técnicas TDD e Test-Last no processo de manutenção corretiva de software**. Orientador: Prof. Me. André Wyzykowski. 2019. 54 f. TCC (Bacharelado em Engenharia de Software) - Universidade Católica do Salvador, Salvador, 2019. Disponível em: <http://ri.ucsal.br:8080/jspui/handle/prefix/879>. Acesso em: 20 abr. 2021.
- DAVI, Tiago. **Elixir: Do zero à concorrência**. 1. ed. Vila Mariana – São Paulo – SP – Brasil: Casa do Código, 2017. 121 p. *E-book*.
- GUSSON, Cássio. **Itaú comete falha com o Pix, transfere R\$1 milhão indevidamente e processa bancos**, 19 jan. 2021. Disponível em: <https://exame.com/invest/itau-comete-falha-com-pix-transfere-1-milhao-indevidament-e-e-processa-bancos/> Acesso em: 23 jan. 2021
- GUEDES, Marylene. **Afinal, o que é TDD?**. Bela Vista - São Paulo - SP: Treinaweb, 4 nov. 2019. Disponível em: <https://www.treinaweb.com.br/blog/afinal-o-que-e-tdd/>. Acesso em: 10 dez. 2020.
- LIMA, Rodrigo Bittencourt de. **Testes de integração automatizados sobre API REST do software Hawa**. Orientador: Dr. Martin Augusto Gagliotti Vigil. 2019. 140 f. TCC (Graduação) - Curso de Sistema de Informação, Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis, 2019. Disponível em: <https://repositorio.ufsc.br/handle/123456789/202488>. Acesso em: 12 dez. 2020.