

FUNDAÇÃO GETULIO VARGAS
ESCOLA DE MATEMÁTICA APLICADA – FGV/EMAp
CURSO DE GRADUAÇÃO EM MATEMÁTICA APLICADA

**Ecossistema Elixir: um exemplo de aplicação para registro de
transações entre contas**

por Rafael Henrique Figueiredo da Purificação

**Rio de Janeiro
2021**

FUNDAÇÃO GETÚLIO VARGAS
ESCOLA DE MATEMÁTICA APLICADA – FGV/EMAp
CURSO DE GRADUAÇÃO EM MATEMÁTICA APLICADA

**Ecossistema Elixir: um exemplo de aplicação para registro de
transações entre contas**

“Declaro ser o único autor do presente projeto de monografia que se refere ao plano de trabalho a ser executado para continuidade da monografia e ressalto que não recorri a qualquer forma de colaboração ou auxílio de terceiros para realizá-lo a não ser nos casos e para os fins autorizados pelo professor orientador.”

Rafael Henrique Figueiredo da Purificação

Orientadora: Asla Medeiros e Sá

Rio de Janeiro
2021

Rio de Janeiro
2021.

Rafael Henrique Figueiredo da Purificação

**Ecossistema Elixir: um exemplo de aplicação para registro de
transações entre contas**

“Trabalho de Conclusão apresentado à Escola de Matemática Aplicada como
requisito para a obtenção parcial do grau de bacharel em Matemática Aplicada.”

Aprovado em ____ de ____ de ____.

Grau atribuído ao Trabalho de Conclusão: ____.

Professora Orientadora: Asla Medeiros e Sá
Escola de Matemática Aplicada - FGV/EMAp
Fundação Getulio Vargas

RAFAEL HENRIQUE FIGUEIREDO DA PURIFICAÇÃO

**“ECOSSISTEMA ELIXIR: UM EXEMPLO DE APLICAÇÃO PARA REGISTRO DE
TRANSAÇÕES ENTRE CONTAS”**

Trabalho de Conclusão de Curso - TCC apresentado ao Curso de Graduação em Matemática Aplicada da Escola de Matemática Aplicada para obtenção do grau de Bacharel (a) em Matemática Aplicada.

Data da Defesa: 09/07/2021

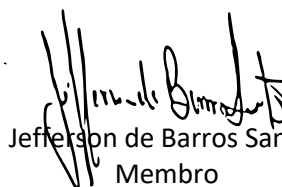
ASSINATURA DOS MEMBROS DA BANCA EXAMINADORA



Asla Medeiros e Sá
Orientadora



Paulo Cezar Pinto Carvalho
Membro



Jefferson de Barros Santos
Membro

Nos termos da Lei nº 13.979 de 06/02/20 - DOU nº 27 de 07/02/20 e Portaria MEC nº 544 de 16/06/20 - DOU nº 114 de 17/06/20 que dispõem sobre a suspensão temporária das atividades acadêmicas presenciais e a utilização de recursos tecnológicos face ao COVID-19, as apresentações dos Trabalhos de Conclusão de Curso, de forma excepcional, serão realizadas de forma remota e síncrona, incluindo-se nessa modalidade membros da banca e discente.

AGRADECIMENTOS

Primeiramente gostaria de agradecer aos meus familiares, principalmente meus pais e meu irmão que sempre me apoiaram nas decisões e que me deram força para concluir o curso e sempre estiveram presentes tanto nos momentos bons, quanto nos ruins.

Aos meus amigos que sempre me ajudaram a extravazar quando necessário e aos meus colegas de curso que sempre me ajudaram a estudar e sempre deram suporte perante aos percalços do período, para isto deixo meu agradecimento especial à Bira, Muniz e Borghi.

Além disso agradeço a todo corpo docente e funcionários da EMAp que sempre se mostraram disponíveis e pacientes para ajudar os alunos. Dentre esses gostaria de fazer um agradecimento especial à minha orientadora, Asla Medeiros e Sá, pelo apoio durante o desenvolvimento do projeto.

Conteúdo

1	Introdução	5
1.1	Experiência e Motivação	5
1.2	Elixir	5
1.2.1	Máquina Virtual do Erlang	7
1.2.2	Ferramentas de desenvolvimento	8
2	Desenvolvimento	12
2.1	Ambiente	12
2.2	Base de Dados	14
2.3	Roteador	18
2.4	Controlador	20
2.5	Funções Principais	22
2.6	Ambiente de Teste	27
3	Conclusão	31
3.1	Aprendizado	31
4	Referências	33

1 Introdução

1.1 Experiência e Motivação

A ideia inicial para desenvolvimento desse projeto veio junto da minha entrada na empresa que me encontro atualmente e a necessidade de ter um conhecimento mais aprofundado em Elixir que é a linguagem utilizada na empresa. Além disso do aprendizado sobre a linguagem também haverá um estudo e demonstração de como criar um sistema financeiro simples, utilizando somente a linguagem supracitada.

Ao longo do texto serão apresentados os benefícios de se trabalhar com Elixir, uma linguagem funcional brasileira que é relativamente nova no meio de programação. Com isso será mostrado o aprendizado obtido e em que tipo de aplicações faz sentido a utilização desse tipo de linguagem.

Após o domínio dos conhecimentos básicos de Elixir, será apresentada um exemplo de aplicação prática da linguagem com um projeto back-end de sistema financeiro, por meio de API, feito em Elixir simulando ações comuns bancárias como cadastro de funcionários, transferências dentro do mesmo sistema, depósito e saque.

1.2 Elixir

Elixir é uma linguagem de programação brasileira, *open-source*, criada em 2012 por José Valim¹, que tinha a intenção de aumentar a extensibilidade e produtividade do Erlang², mantendo a compatibilidade com ferramentas e ecossistemas desse, uma vez que trariam benefícios a linguagem como funcionalidade, tolerância a falhas e principalmente concorrência. As maiores diferenças entre Erlang e Elixir vêm das ferramentas que são implementadas em Elixir e que trazem muitos benefícios para a linguagem, como o *framework web* Phoenix, além disso Elixir é uma linguagem de

¹Elixir (linguagem de programação) - Wikipédia

²<https://www.erlang.org/>

fácil manutenção e debug, além de possuir uma facilidade de escalabilidade, diferente do Erlang.

Essa linguagem foi um projeto da Plataformatec, empresa que José Valim foi cofundador. Plataformatec foi fundado por Valim e alguns amigos que ao participar de várias conferências brasileiras sobre Ruby³, uma linguagem de programação muito conhecida no desenvolvimento de software, tornaram-se importantes membros na comunidade da linguagem. Valim ainda participou do desenvolvimento do *Ruby on Rails*⁴, um *framework* para aplicações *web* que utiliza a linguagem de programação Ruby. As duas ferramentas tiveram como principal intenção deixar o trabalho do programador mais produtivo e como o próprio site do Ruby diz: "O melhor amigo do programador".

José Valim fez a escolha de criar uma nova linguagem, porque mesmo tendo muita proximidade com Ruby e ter feito parte do time principal de desenvolvimento do Ruby on Rails e apreciando a flexibilidade, fluidez e elegância da sintaxe, sabia que Ruby teria limitações dependendo da quantidade de dados transferidos e prejudicaria a velocidade e processamento de envio dessas informações. Por isso escolheu a máquina virtual do Erlang para executar o Elixir, que além de possuir todas as características do Ruby, como o paradigma funcional, o código aberto, tem também aplicações distribuídas com baixa latência e é tolerante a falhas. Fazendo o Elixir ser funcional por definição e também possuir imutabilidade pra resolver principalmente os problemas de concorrência que percebia usando Ruby.

Depois de falar um pouco sobre a história do Elixir e entender um pouco da criação dessa linguagem, vamos falar dos elementos que tornam essa linguagem ideal para o projeto final e dar alguns exemplos de usabilidade.

³Segundo a RedMonk, uma empresa de analistas do setor focada em desenvolvedores de software, Ruby está entre uma das 10 linguagem mais usadas no mundo.

⁴Ruby on Rails Website

1.2.1 Máquina Virtual do Erlang

Erlang é uma linguagem de programação e também um ambiente de execução. Erlang tem suporte nativo para concorrência, distribuição de aplicações e tolerância a falhas. É uma linguagem usada geralmente para sistemas de telecomunicação da *Ericsson*. A máquina virtual do Erlang, também conhecida como BEAM (Bogdan/Björn's Erlang Abstract Machine)⁵, também possui essas qualidades e por isso foi escolhida para rodar o ambiente de execução do Elixir.

A BEAM é a responsável por executar o código compilado do Erlang e do Elixir que traz benefícios para o Elixir. Para entender melhor os benefícios da BEAM, devemos entender o que são processos e como os processos se comunicam nesse ambiente.

Um processo na BEAM é um contexto de execução do *runtime* do código. Os benefícios de se trabalhar nessa máquina virtual é que os processos conseguem se comunicar por meio de mensagens que avisam assim que um processo é iniciado e finalizado, tornando esses processos independentes uns dos outros, onde os mesmos utilizam espaços de memória, fluxos de execução, assim os tornando em dois programas separados.

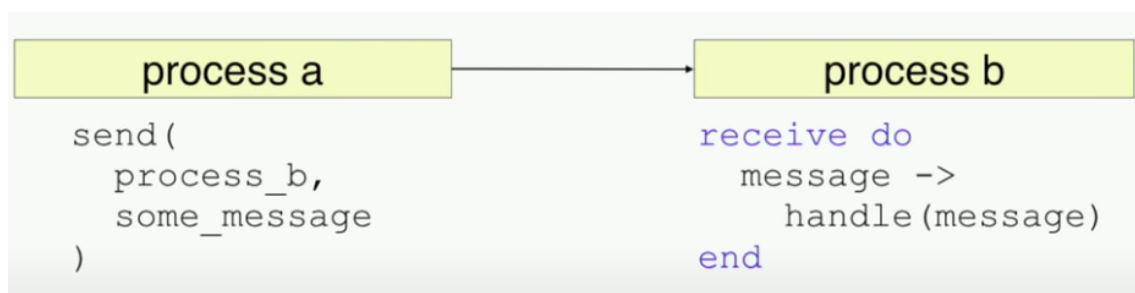


Figura 1: Comunicação de Processos

Uma mesma função pode ser dividida em mais de um processo, isso faz com que, caso uma parte da função dê errado, isso não prejudique todos os processos que estão rodando concorrentemente, ou seja, a aplicação não trava se apenas uma

⁵Erlang Site

parte estiver dando problema, pois estão rodando em lugares separados.

1.2.2 Ferramentas de desenvolvimento

Além das funcionalidades que a linguagem já tem por padrão por efeito do uso da BEAM, temos ferramentas que fazem parte do dia a dia do desenvolvimento e que são muito importantes para o desenvolvimento Elixir⁶. Dentre elas temos:

Mix é uma ferramenta de construção que ajuda a criar, compilar e testar a aplicação, além de ajudar a gerenciar as dependências necessárias para execução da mesma.

Iex que significa *Interactive Elixir*, é um *Shell* de execução, onde se pode executar expressões e até algumas expressões básicas.

Hex é o gerenciador de pacotes para todo o ecossistema da BEAM.

ExUnit é o framework de testes unitários, que auxilia na execução dos testes da aplicação, em conjunto com o Mix fica ainda mais poderosa.

ExDoc é como o Elixir gera toda a sua documentação, é feita toda em *markdown* e é muito útil em toda atividade de desenvolvimento, principalmente quando se está trabalhando em algum editor de texto, que ajuda você acessar a documentação da função.

Phoenix é um framework de desenvolvimento web escrita em Elixir e que implementa o padrão *Model View Controller*(MVC) do lado do servidor. O MVC é um padrão de arquitetura de software focado no reuso de código e a separação de conceitos em três camadas interconectadas, onde a apresentação dos dados e interação dos usuários(front-end) são separados dos métodos que interagem com o banco de dados(back-end).⁷

Ecto é a ferramenta que trata e faz a interação dos dados com os banco de dados.

⁶Todas as ferramentas tiveram referência no HexDocs, um site que documenta todas as ferramentas e pacotes de elixir

⁷Hopkins, Callum (4 de março de 2013). «The MVC Pattern and PHP» [O padrão MVC e o PHP]. SitePoint Pty.

Dentre essas ferramentas, algumas têm ainda detalhes que precisam ser melhor explicitados para que se consiga desenvolver um projeto completo em Elixir. Então antes de começar o desenvolvimento vamos ainda levantar alguns pontos importantes sobre o Phoenix e o Ecto.

1.2.2.1 Phoenix Para rodar o Phoenix temos que ter algumas dependências instaladas, dentre elas precisamos ter o Erlang VM (BEAM), o Elixir, alguma base de dados como SQL Server ou PostgreSQL. Há também opções de base de dados NoSQL (não relacionais), mas há poucas implementações de drivers e por isso são pouco usado em Elixir. Dentro do Phoenix temos algumas ferramentas muito importantes para o desenvolvimento que iremos utilizar no nosso projeto, dentre elas temos:

Phoenix.Controller é o que recebe a conexão e os parâmetros para executar a função principal do código. Dessa maneira podemos controlar o que retornar para a aplicação web e também mudar os parâmetros recebidos da web para a função local rodar da melhor forma.

```
defmodule MyAppWeb.UserController do
  use MyAppWeb, :controller

  def show(conn, %{"id" => id}) do
    user = Repo.get(User, id)
    render(conn, "show.html", user: user)
  end
end
```

Figura 2: Exemplo de um *Phoenix.Controller*

Phoenix.View é um módulo usado para definir a visualização principal de uma aplicação, serve de base para fazer templates que serão renderizados pelo controller para a aplicação web.

```
defmodule YourApp.UserView do
  use YourApp.View

  def render("show.json", %{user: user}) do
    %{name: user.name, address: user.address}
  end
end
```

Figura 3: Exemplo de um *Phoenix.View*

Além dessas há também outras ferramentas que são muito úteis como o *Channel*, *Router* e *Logger*, mas que não serão usados no projeto porque nesse caso não traria ganhos efetivos para o projeto, mas são muito úteis em projetos maiores facilitando bem a organização do código e a interação entre os programas e com a interface final, caso tivesse.

1.2.2.2 Ecto O Ecto também oferece diversas facilidades para a criação e adaptação de bases de dados. O Ecto pode ser dividido em 4 componentes principais, *Repo*, *Schema*, *Changeset* e *Query*.

Ecto.Repo repositórios são invólucros do armazenamento dos dados. Através do repositório, podemos criar, atualizar, excluir e consultar uma base de dados. Um repositório precisa de um adaptador e credenciais para se comunicar com o banco de dados.

Ecto.Schema são usados para mapear os dados para uma estrutura compatível com Elixir. Normalmente é usado para mapear também tabelas, mas não é necessário usar Ecto.

Ecto.Changeset é uma ferramenta para ajudar os desenvolvedores a filtrar e moldar parâmetros externos, também ajudam a validar mudanças e se as entradas estão correspondentes com as especificações da base de dados.

Ecto.Query são escritas em Elixir e são usadas para obter informações da base de dados e são mais seguras do que usar SQL *queries* e conseguem ter uma fle-

xibilidade maior no código deixando os desenvolvedores contruíram as *queries* de maneira separada ao invés de uma vez só.

```
defmodule User do
  use Ecto.Schema

  import Ecto.Changeset

  schema "users" do
    field :name
    field :email
    field :age, :integer
  end

  def changeset(user, params \\ %{}) do
    user
    |> cast(params, [:name, :email, :age])
    |> validate_required([:name, :email])
    |> validate_format(:email, ~r/@/)
    |> validate_inclusion(:age, 18..100)
  end
end
```

Figura 4: Exemplo de um *Ecto's Schema* e *Changeset*

2 Desenvolvimento

Depois de falar sobre Elixir e algumas ferramentas próprias do ecossistema de desenvolvimento web envolvendo Phoenix, Ecto e a máquina virtual do Erlang (BEAM), vamos começar o desenvolvimento do projeto de criação de uma API de banking que simula algumas interações básicas de uma interface web com um banco de dados, tais elas como: a criação de uma conta, transferência entre contas, depósito e saque.

Para isso iremos usar um editor de texto de código chamado *Visual Studio Code* (*VSCode*). Além dele também utilizaremos o *Docker*⁸, que é uma ferramenta de virtualização de aplicações utilizando *containers* que formam uma imagem completa de todas as dependências necessárias para executar a aplicação. No caso da nossa aplicação, a única imagem que vai ser construída é a do *PostgreSQL*, a base de dados que optei em usar. A utilização do *Docker* deixa mais fácil porque ao executar uma aplicação não se faz mais necessário ter as dependências instaladas na máquina e mais rápido porque não precisa executar máquinas virtuais para cada aplicação que quiser rodar. Também será utilizado o *Docker Compose* que é uma ferramenta para definir e rodar múltiplos containers de uma aplicação.

É necessário ter instalado os seguintes programas na máquina: ***Docker***, ***Docker Compose***, ***Erlang*** e ***Elixir***.

2.1 Ambiente

Após instalar todos os programas necessários, devemos começar a desenvolver o código da aplicação em si. Para isso iremos utilizar o *Mix* que já foi mencionado acima como a nossa ferramenta de construção e compilação do código. Como vamos usar *Phoenix* para desenvolver temos como criar um template básico utilizando um comando mix:

⁸<https://www.docker.com/company>

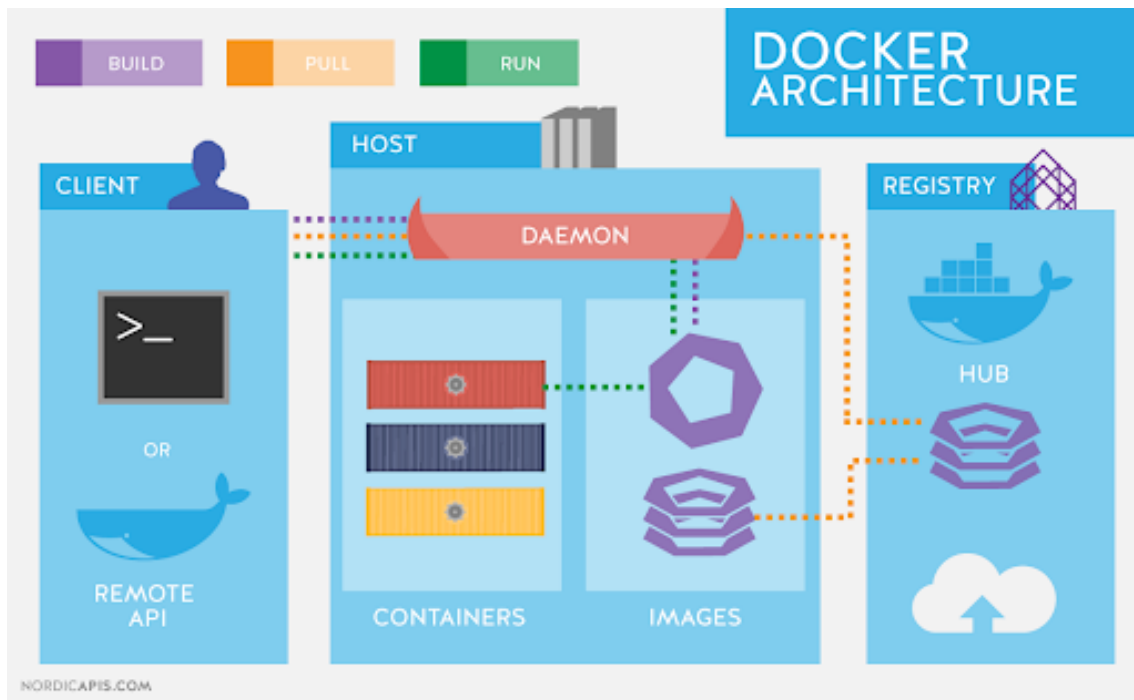


Figura 5: Arquitetura do *Docker*

```
$ mix phx.new api_banking_challenge
```

Só com essa linha de código são criadas todas as pastas principais necessárias para a organização do código, as dependências principais para execução do Ecto e do Phoenix e os principais arquivos de execução da aplicação.

Com isso, basicamente a estrutura completa do código está montada. Para continuar o desenvolvimento do projeto é importante que nossa estrutura de dados seja pensada previamente. Para isso foi pensado numa estrutura onde haveriam duas bases de dados, uma responsável por guardar os dados dos clientes e a outra pelas transações.

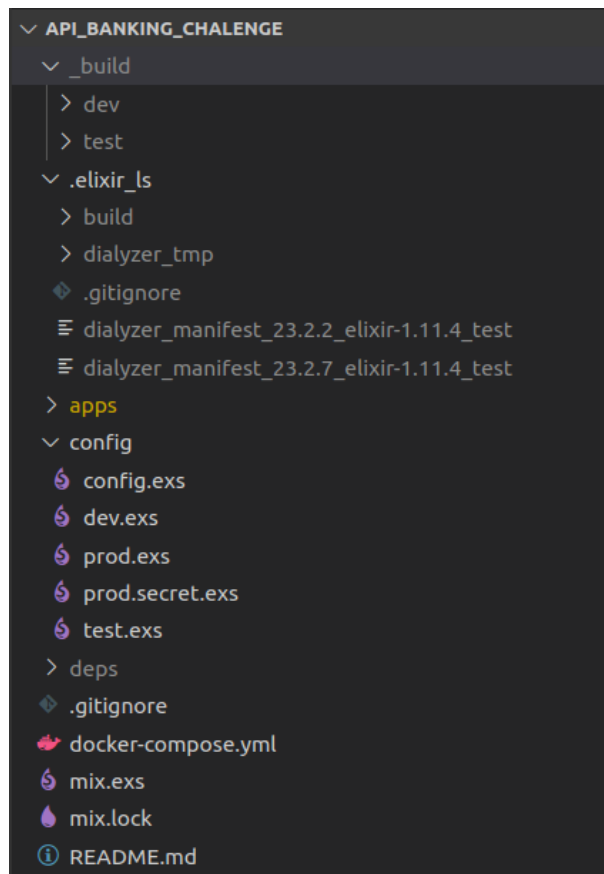


Figura 6: Estrutura do código gerada pelo Mix

2.2 Base de Dados

Antes de tudo foi necessário pensar como seria a estrutura dos dados e como as bases se comunicariam. Pra isso foi pensado na criação de 2 tabelas que teriam uma estrutura igual a apresentada na figura abaixo.

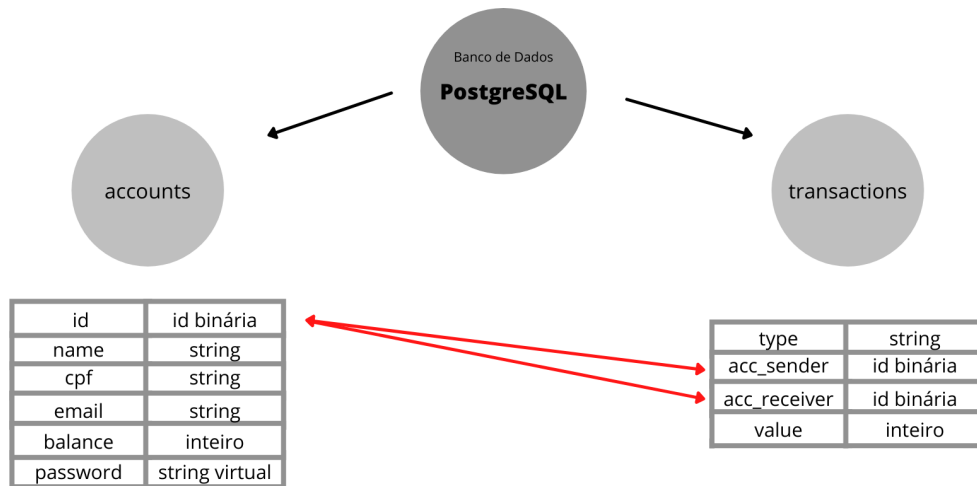


Figura 7: Estrutura do banco de dados que vai ser utilizado

A primeira tabela criada foi a de contas, chamada no projeto de *account.ex*. Essa base terá 5 campos de entrada e 1 campo autogerado, a seguir será mostrado como o código foi feito e os campos que cada um representa.

```
defmodule ApiBankingChallenge.Schemas.Account do
  @moduledoc """
  The entity of accounts

  """

  use Ecto.Schema

  import Ecto.Changeset

  @possible_params [:name, :cpf, :email, :password]

  @primary_key {:id, :binary_id, autogenerate: true}
  @foreign_key_type :binary_id
  schema "accounts" do
```

```

    field(:name, :string)
    field(:cpf, :string)
    field(:email, :string)
    field(:balance, :integer)
    field(:password, :string, virtual: true)

    timestamps()
end

def changeset(model \\ %__MODULE__{}, params) do
  model
  |> cast(params, @possible_params)
end
end

```

Listing 1: account.ex

O *@moduledoc* é responsável por guardar as informações sobre o módulo apresentado. Desse jeito, em qualquer lugar que o módulo for usado será possível saber as informações que estão contidas nessa variável. Usamos também o *Ecto.Schema* e o *Ecto.Changeset* para, respectivamente, criar a base de dados e fazer a verificação dos campos que foram passados como entrada. As colunas criadas foram *:name*, *:cpf*, *:email*, *:balance* e *:password*.

Similar a tabela de contas, temos a tabela de transações. Nessa tabela há uma diferença onde validamos antes de inserir na base a inclusão de um tipo específico de tipo de transação e também validamos os campos que são necessários para a inclusão na base de dados.

```

defmodule ApiBankingChallenge.Schemas.Transaction do
  @moduledoc """
  The entity of transitions

  """

```

```

use Ecto.Schema
import Ecto.Changeset

@required_params [:value]
#sender and receiver are optional because that's how we do the
  deposit and withdraw
#operations
@optional_params [:acc_sender_id, :acc_receiver_id]

@primary_key {:id, :binary_id, autogenerate: true}
@foreign_key_type :binary_id

schema "transactions" do
  field(:type, :string)
  field(:acc_sender_id, :binary_id)
  field(:acc_receiver_id, :binary_id)
  field(:value, :integer)

  timestamps()
end

def changeset(model \\ %__MODULE__{}, params) do
  model
  |> cast(params, @required_params ++ @optional_params)
  |> validate_required(@required_params)
  |> validate_inclusion(:type, ["transaction", "withdraw", "
    deposit"])
end
end

```

Listing 2: transaction.ex

A função *validate_required* valida todos os campos que são requeridos pela base de dados, os campos requeridos são ajustáveis com a variável *@required_params*. E com a função *validate_inclusion* nós validamos se nesse campo *:type* tem contido nele um dos tipos de transação predefinidos, entre eles: uma transação entre duas contas, saque e depósito.

2.3 Roteador

O roteador(*Phoenix.Router*) é onde direcionamos as requisições HTTP para dentro do Elixir e executamos as funções no nosso código principal. Para essa aplicação teremos três requisições relacionado às contas, que são a criação das contas, exposição dos dados de uma conta e a exclusão desses dados. Já as requisições para transações são relacionadas aos três tipos de transações, sendo eles transação, depósito e saque.

```
defmodule ApiBankingChallengeWeb.Router do
  use ApiBankingChallengeWeb, :router

  pipeline :api do
    plug(:accepts, ["json"])
  end

  scope "/api", ApiBankingChallengeWeb do
    pipe_through(:api)

    # accounts
    post("/accounts", AccountsController, :create)
    get("/accounts/:id", AccountsController, :show)
    delete("/accounts/:id", AccountsController, :delete)

    # transactions
    post("/transactions", TransactionsController, :transaction)
    post("/transactions", TransactionsController, :withdraw)
    post("/transactions", TransactionsController, :deposit)
  end
end
```

Listing 3: router.ex

Aqui pode-se observar o roteador chamando as requisições HTTP(post, get e delete) e chamando as respectivas funções. O método POST é utilizado para submeter uma entidade a um recurso específico, frequentemente causando uma mudança no estado do recurso ou efeitos colaterais no servidor, já o método GET solicita a representação de um recurso específico, essas requisições retornam somente dados e por último temos a requisição DELETE que tem como função remover um recurso específico.

2.4 Controlador

Como já falado antes o *Phoenix.Controller* recebe a requisição do roteador e os parâmetros para execução de uma função. É nessa etapa que é configurada a resposta dada para o servidor, tanto em caso de sucesso, quanto em caso de falha de execução da função.

```
defmodule ApiBankingChallengeWeb.AccountsController do
  @moduledoc """
    Web layer for the Account resource.
  """

  use ApiBankingChallengeWeb, :controller

  alias ApiBankingChallenge.Accounts

  def create(conn, params) do
    params
    |> Accounts.create()
    |> case do
      {:ok, account} ->
        send_json(conn, 201, "account.json", account: account)

      {:error, error} ->
        case error do
          :email_conflict ->
            send_json(conn, 400, "already_taken.json", key: "
              Email")

          :cpf_conflict ->
            send_json(conn, 400, "already_taken.json", key: "CPF"
              )

          :invalid_info ->
```

```

        send_json(conn, 400, "invalid_info.json")
    end
end
end

def show(conn, %{"id" => id}) do
  id
  |> Accounts.get()
  |> case do
    {:ok, account} ->
      send_json(conn, 200, "account.json", account: account)

    {:error, :invalid_info} ->
      send_json(conn, 400, "invalid_info.json")

    {:error, :not_found} ->
      send_json(conn, 400, "not_found.json")
  end
end

def delete(conn, %{"id" => id}) do
  id
  |> Accounts.delete()
  |> case do
    {:ok, account} ->
      send_json(conn, 200, "account.json", account: account)

    {:error, :invalid_info} ->
      send_json(conn, 400, "invalid_info.json")

    {:error, :not_found} ->
      send_json(conn, 400, "not_found.json")
  end
end

```

```

    end
end

defp send_json(conn, status, view, opts \\ %{}) do
  conn
  |> put_status(status)
  |> render(view, opts)
end
end

```

Listing 4: `accounts_controller.ex`

Apresentado acima, está um exemplo de um *Phoenix.Controller*. Podemos observar que para cada função encontrada aqui é executada a função da aplicação e com o retorno esperado é feito o tratamento da resposta. Em caso de sucesso é retornado um status 201 ou 200 que representa sucesso da operação, uma string contendo um arquivo no formato *JSON* e a conta em questão, caso tenha algum problema, cada erro é tratado de uma forma. Como um exemplo, ao informarmos um CPF de maneira equivocada, será informado um erro do *:cpf_conflict* que resultará no envio de uma mensagem de erro no formato de um *already_taken.json*. Essas mensagens de erro são encontradas na pasta *views* utilizando uma ferramenta do *Phoenix* já apresentado anteriormente.

De forma análoga está a parte de transações, mas ao invés de serem essas funções mostradas acima, encontramos as funções que já foram definidas no roteador.

2.5 Funções Principais

Nesta etapa será feita a conexão com o banco de dados e a requisição de criação, exclusão ou obtenção de informações da base de dados. Para acessar a base de dados das contas é utilizado o *Ecto.Repo*. Para a função de criação utilizamos ainda uma outra função do *Ecto* para poder validar as entradas e garantir que estão sendo inseridos somente dados no formato correto. Para isso utilizamos o *Ecto.Changeset*.

```

def changeset(model \\ %__MODULE__{}, params) do

```



```

model
|> cast(params, @required_params)
|> validate_required(@required_params)
|> validate_length(:password, min: 8)
|> validate_format(:cpf, ~r/^\d{3}\.\d{3}\.\d{3}\-\d{2}$/)
|> validate_format(:email, ~r/^[^@ \t\r\n]+@[^@ \t\r\n]+\.[^@
\t\r\n]+/)

```

Listing 5: create.exs - changeset

Como se apresenta acima, temos um modelo que valida o tamanho da senha enviada, valida o formato do CPF utilizando um modelo de *Regex* e também faz o mesmo para verificar o email.

```

def create(params) do
  # Validating input
  with %{valid?: true, changes: changes} <- Create.changeset(
    params),
    # Checking if the account doesn't exist and hashing it's
    # password.
    %{valid?: true} = unique <- Account.changeset(changes),
    # Inserting into the repository.
    {:ok, account} <- Repo.insert(unique) do
    {:ok, account}
  else
    %{valid?: false} ->
      {:error, :invalid_info}
  end
rescue
  error in Ecto.ConstraintError ->
    case error do
      %{constraint: "accounts_cpf_index"} ->
        {:error, :cpf_conflict}
    end
  _ ->
    {:error, :unknown_error}
end

```

```

        %{constraint: "accounts_email_index"} ->
            {:error, :email_conflict}
    end
end

def get(account_id) do
    case Ecto.UUID.cast(account_id) do
        {:ok, _} ->
            case Repo.get(Account, account_id) do
                nil -> {:error, :not_found}
                account -> {:ok, account}
            end

            :error ->
                {:error, :invalid_info}
        end
    end
end

def delete(account_id) do
    case get(account_id) do
        {:ok, account} -> Repo.delete(account)
        {:error, error} -> {:error, error}
    end
end
end

```

Listing 6: accounts.ex

Aqui é possível ver a inserção dos dados por meio do *Repo* e também as mensagens de erro que aparecem no controlador, mostrando assim como os erros são apontados. Para mostrar melhor como as transações funcionam será demonstrado como foi feito a função de transação principal.

```

def transaction(%{
  value: value,
  acc_sender_id: sender,
  acc_receiver_id: receiver
}) do
  with _sender <- Ecto.UUID.cast!(sender),
       _receiver <- Ecto.UUID.cast!(receiver),
       # casting sender to get data
       acc_sender <- Repo.get!(Account, sender),
       acc_receiver <- Repo.get!(Account, receiver),
       %{valid?: true} = changeset <-
         Transaction.changeset(%{
           value: value,
           acc_sender_id: acc_sender.id,
           acc_receiver_id: acc_receiver.id,
           type: "transaction"
         }) do
    if acc_sender.balance >= value do
      update_sender_balance =
        Ecto.Changeset.change(acc_sender, balance: acc_sender.
          balance - value)

      update_receiver_balance =
        Ecto.Changeset.change(acc_receiver, balance:
          acc_receiver.balance + value)

      Repo.update(update_sender_balance)
      Repo.update(update_receiver_balance)

      Logger.info("Transfer succeeded from #{acc_sender.name} to
        #{acc_receiver.name}")
      Repo.insert(changeset)
    end
  end
end

```

```

    else
      {:error, :not_enough_funds}
    end
  end
end
rescue
  err in Ecto.CastError ->
    Logger.error(err)
    {:error, :invalid_info}

  err in Ecto.NoResultsError ->
    Logger.error(err)
    {:error, :not_found}
end

```

Listing 7: transactions.ex

A função de transação recebe uma entrada do tipo *map* que representa um dicionário. Nele estão inclusos os valores da transação e os ids do remetente e do destinatário. O primeiro passo é confirmar se os valores são ids no formato desejado e pra isso já utilizamos a função nativa do *Ecto*, *cast!*. Após essa verificação é feito uma requisição para a base de dados de contas para que seja obtido o valor que o remetente possui na conta e seja feita uma análise capaz de confirmar se o valor que o mesmo possui é maior que a quantia desejada para envio. Se essa premissa for satisfeita é feita a alteração do valor entre as duas contas e são atualizadas na base de dados de contas. Além disso é gerada uma mensagem confirmando essa transferência. Em caso de falta de fundos, é feito um tratamento específico retornando um aviso de falta de fundos para o controlador, já em outros casos como erro de valores de entrada ou sem resultado na busca dos ids na base, o próprio *Ecto* tem um jeito de tratar esses erros.

Para as funções de saque e depósito são feitas pequenas alterações, mas a estrutura é bem parecida. Somente mudam os valores de entrada e o tipo de transação que é registrado na base de dados de transação, que também discrimina os dados

que são entrados por tipo de transação.

2.6 Ambiente de Teste

O Elixir tem um ambiente eficiente de testes e com o auxílio do *Mix* o processo de teste fica ainda mais fácil de ser executado. A estrutura dos testes é basicamente a execução do código com os dados de entrada pré definidos e utilizado uma função chamada *assert* que verifica se o retorno da função foi correto. Para entender como um teste funciona será demonstrado a seguir os testes que são realizados para testar o funcionamento da função responsável pela criação de uma conta.

```
describe "create/1" do
  test "create an account with correct data" do
    name = Ecto.UUID.generate()
    email = "#{Ecto.UUID.generate()}@test.com"
    password = Ecto.UUID.generate()
    cpf = "000.000.000-00"

    params = %{
      name: name,
      email: email,
      password: password,
      cpf: cpf
    }

    assert {:ok, account} = Accounts.create(params)

    assert account.name == name
    assert account.email == email
    assert account.cpf == cpf
  end
end
```

```

@tag capture_log: true
test "yields error when CPF is invalid" do
  name = Ecto.UUID.generate()
  email = "#{Ecto.UUID.generate()}@test.com"
  password = Ecto.UUID.generate()
  cpf = "invalid CPF"

  params = %{
    name: name,
    email: email,
    password: password,
    cpf: cpf
  }

  assert {:error, _} = Accounts.create(params)
end

[...]

@tag capture_log: true
test "yields error when CPF is already taken" do
  name = Ecto.UUID.generate()
  email = "#{Ecto.UUID.generate()}@te@st.com"
  password = Ecto.UUID.generate()
  cpf = "000.000.000-00"

  params = %{
    name: name,
    email: email,
    password: password,
    cpf: cpf
  }

```

```

Accounts.create(%{
  name: name,
  email: "other@email.com",
  password: password,
  cpf: cpf
})

assert {:error, :cpf_conflict} = Accounts.create(params)
end
end

```

Listing 8: *accounts_test.ex*

O primeiro teste é relacionado ao sucesso da criação da conta. Com os dados sendo enviados de maneira correta, o resultado esperado seria uma mensagem de sucesso vindo da função. Com isso é feito uma confirmação de que o retorno da função estaria no formato `{:ok, account}`, onde o `:ok` significa que a execução da função foi um sucesso e *account* recebe os dados de retorno da função. A seguir é feita uma comparação se o retorno é igual às informações passadas a função. O próximo teste exige que ao passar um CPF inválido, nesse caso uma string que não se comporta conforme o *Regex* que foi passado, o retorno seja um erro. No outro teste fazemos o registro de duas contas na base de dados e verificamos que o CPF está duplicado, nesse esperamos o erro *:cpf_conflict*.

Importante ressaltar que esses são apenas alguns exemplos de testes que são realizados. No total, foram criados vinte e quatro testes que estressam as possibilidades de erro do código. A execução desses testes é feito através de um comando simples do *Mix* e roda todos os testes relacionados da aplicação.

```
$ mix test
```

O retorno da execução dessa linha de comando são os logs feitos pelo Ecto e o resultado dos testes executados. Nesse caso são apresentados diversas linhas de

erro porque utilizamos alguns erros nativos do *Ecto*. Com isso podemos ver que a tratativa de erros dessa ferramenta é extremamente poderosa e consegue ir nos detalhes e apresentar uma verificação de erros bem específica para o usuário final deixando bem fácil o processo de depuração do código.

```
rafael@rafael-stn:~/Projects/Stone/api_banking_challenge$ mix test
==> api_banking_challenge
Compiling 1 file (.ex)
Generated api_banking_challenge app
==> api_banking_challenge
...12:44:53.021 [error] [_exception__: true, __struct__: Ecto.NoResultsError, message: "expected at least one result but got none in query:\n\nfrom a0 in ApiBankingChallenge.Schemas.Account,\n where: a0.id == ^\"59ee2caa-d786-4725-949c-2326c2931d6a\"\\n"]
..12:44:53.029 [error] [_exception__: true, __struct__: Ecto.NoResultsError, message: "expected at least one result but got none in query:\n\nfrom a0 in ApiBankingChallenge.Schemas.Account,\n where: a0.id == ^\"aef79266-bd31-493c-a6d1-fe335832b445\"\\n"]
.....12:44:53.042 [error] [_exception__: true, __struct__: Ecto.CastError, message: "cannot cast \"invalid id\" to Ecto.UUID", type: Ecto.UUID, value: "invalid id"]
...12:44:53.054 [error] [_exception__: true, __struct__: Ecto.CastError, message: "cannot cast \"invalid id\" to Ecto.UUID", type: Ecto.UUID, value: "invalid id"]
..12:44:53.055 [error] [_exception__: true, __struct__: Ecto.NoResultsError, message: "expected at least one result but got none in query:\n\nfrom a0 in ApiBankingChallenge.Schemas.Account,\n where: a0.id == ^\"bb42bf10-fcd8-4d76-9c32-80cebbf151e1\"\\n"]
.12:44:53.055 [error] [_exception__: true, __struct__: Ecto.CastError, message: "cannot cast \"invalid id\" to Ecto.UUID", type: Ecto.UUID, value: "invalid id"]
.
Finished in 0.1 seconds
22 tests, 0 failures

Randomized with seed 863185
==> api_banking_challenge_web
..
Finished in 0.03 seconds
2 tests, 0 failures

Randomized with seed 863185
```

Figura 8: Retorno exibido no terminal pelo mix.test

Após a conclusão dos testes e perceber que todos passaram, temos um modelo funcional de operação bancária constituído de um sistema *back-end* que constrói base de dados de contas e transações e faz as operações de envio, retirada e depósito nas contas. Para o modelo estar em funcionamento precisamos rodar apenas mais três comandos que iniciarão as ferramentas e o servidor.

O primeiro comando que necessita ser executado é o do *Docker Compose* que iniciará a containerização do *PostgreSQL* que nesse caso é a única imagem que será utilizada. O arquivo de execução do *Docker Compose* é criado automaticamente pelo Mix na criação do projeto.


```
$ docker-compose up -d
```

Após essa execução é preciso rodar três comandos do *Mix* que compreendem em baixar as dependências, fazer o setup inicial da aplicação e iniciar o servidor do *Phoenix*.

```
$ mix deps.get
```

```
$ mix setup
```

```
$ mix phx.server
```

Com o servidor rodando, agora só se faz necessária a criação de uma interface que interaja com a API já criada para que assim, os comandos criados possam ser rodados. Os arquivos desse projeto já estão disponibilizados no GitHub⁹.

3 Conclusão

Esse projeto teve como ideia principal apresentar a linguagem Elixir e os benefícios de sua utilização. Para exemplificar uma utilização prática da linguagem foi feita uma API onde foi demonstrado desde a organização do projeto até a execução final e inicialização do servidor para funcionamento da API. Com isso foi desenvolvido uma API de criação de contas, onde também é possível realizar operações entre as mesmas. Foi um projeto onde tentei ao mesmo tempo que aprendia a linguagem e suas implementações, também entender sobre o negócio da empresa. Tentei apresentar por meio desse documento o passo a passo que tive sobre a parte prática de utilização do Elixir e levantar também pontos que considerei importantes ao estudar melhor a linguagem.

3.1 Aprendizado

Esse projeto foi importante para botar em prática diversos conhecimentos de estrutura de dados. Foi a primeira atividade realizada inteiramente por mim e

⁹Projeto api banking challenge no GitHub

que teve um ganho muito alto de conhecimento em aplicações web. Além disso foi possível trabalhar um pouco da habilidade com a escrita, que creio ter sido um dos maiores desafios desse trabalho final.

Além disso também é um projeto que serviu de iniciação da linguagem Elixir, que mesmo sendo bem recente está sendo amplamente usada por diversas empresas, principalmente as que estão iniciando no mercado agora. Dentre as empresas que usam, estão: Cabify, BBC, Discord, Pepsi Co, Go Daddy, entre outras. Ou seja, foi um processo que rendeu muitos aprendizados e fez introdução a essa linguagem incrível que me fez entender mais sobre a estrutura de uma linguagem de programação e ter um pensamento mais crítico em relação a qual linguagem usar para determinada atividade.

4 Referências

Oliveira; Fraga; Montez. Programação em Sistemas Distribuídos. Escola de Informática da SBC-Sul - ERI 2002.

Documentação do Erlang. Site

Documentação do Elixir. Site

Documentação do Hex (Hex Docs). Site

Jurić, Saša (2019). Elixir in Action.

Valim, José (2016). Programming Phoenix: Productive —, Reliable —, Fast

Valim; Tate(2018). Adopting Elixir: From Concept to Production

The RedMonk Programming Language Rankings: January 2021