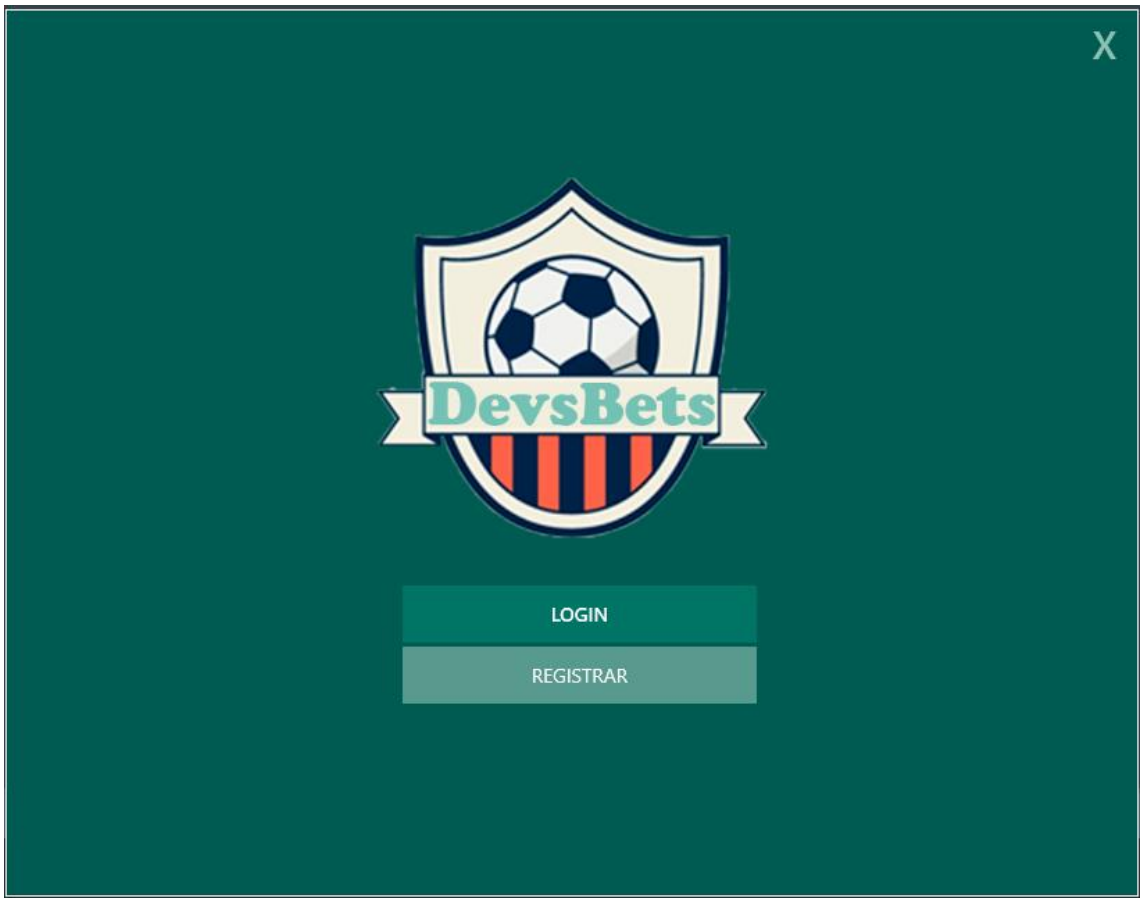
DevsBets

(Recursos Técnicos Utilizados)



Sumário

1. Padrões de Projeto

1.1. Singleton

Foi utilizado o padrão Singleton para armazenar uma instância única para o usuário logado na aplicação. Este padrão foi desenvolvido na classe ***TUserAuthenticated*** da unit ***UService.User.Authenticated.pas***

```
type
  //Classe utilizando o Padrão Singleton
  //Para armazenar o usuário logado
  TUserAuthenticated = class
    private
      FUser: TUser;

      function GetUser: TUser;
      procedure SetUser(const Value: TUser);
    public
      constructor Create;
      destructor  Destroy; override;

      class function GetInstance: TUserAuthenticated;
      class function NewInstance: TObject; override;

      property User: TUser read GetUser write SetUser;
  end;

var
  GbInstance: TUserAuthenticated;
```

Figura 1 - Singleton

1.2. Prototype

Foi utilizado o padrão Prototype para criar uma cópia idêntica do time no cadastro das partidas com o intuito de criarmos menos código.

```
function TTeam.Clone: TTeam;
begin
  Result := TTeam.Create;

  Result.FId   := Self.FId;
  Result.FName := Self.FName
end;
```

Figura 2 - UEntity.Teams

```
procedure TfraMatchRegistry.Registrar;
var
  xServiceMatch: IService;
  xHora: TTime;
  xData: TDate;
  xTimeAux: TTeam;
  xTimeA, xTimeB: TTeam;
begin
  if Trim(edtHora.Text) = EmptyStr then
    raise Exception.Create('Informe a Hora da Partida.');

  if Trim(edtData.Text) = EmptyStr then
    raise Exception.Create('Informe a Data da Partida.');

  if cmbTimeA.ItemIndex = -1 then
    raise Exception.Create('Informe o Time A da Partida.');

  if cmbTimeB.ItemIndex = -1 then
    raise Exception.Create('Informe o Time B da Partida.');

  if cmbTimeA.ItemIndex = cmbTimeB.ItemIndex then
    raise Exception.Create('Informe Times diferentes para a Partida.');

  xHora  := StrToTime(Trim(edtHora.Text));
  xData  := StrToDate(Trim(edtData.Text));

  xTimeAux := TTeam(cmbTimeA.Items.Objects[cmbTimeA.ItemIndex]);
  xTimeA   := xTimeAux.Clone;

  xTimeAux := TTeam(cmbTimeB.Items.Objects[cmbTimeB.ItemIndex]);
  xTimeB   := xTimeAux.Clone;

  xServiceMatch := TServiceMatch.Create(
    TMatch.Create(xData, xHora, xTimeA, xTimeB));

  xServiceMatch.Registrar;
  Self.VoltarTela;
```

Figura 3 - UFraMatch.Registry

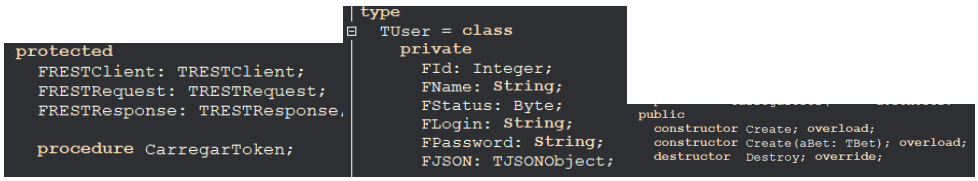
2. Recursos na Programação

2.1. Orientação a Objetos

Foram utilizadas diversas classes, cada uma seguindo o seu propósito, ou seja, o seu domínio de assunto. Na nossa aplicação temos os seguintes domínios de assuntos: *Users, Teams, Matchs, Bets e Login*. Cada um desses domínios possui suas camadas de negócio, controladores e telas.

2.2. Encapsulamento

Foram utilizados basicamente três tipos de encapsulamento: *Private, Protected e Public.*



2.3. Generics

Um dos recursos novos a partir da versão 2009/2010 do Delphi é o Generics. Utilizamos ele para criarmos uma lista de objetos genérico.

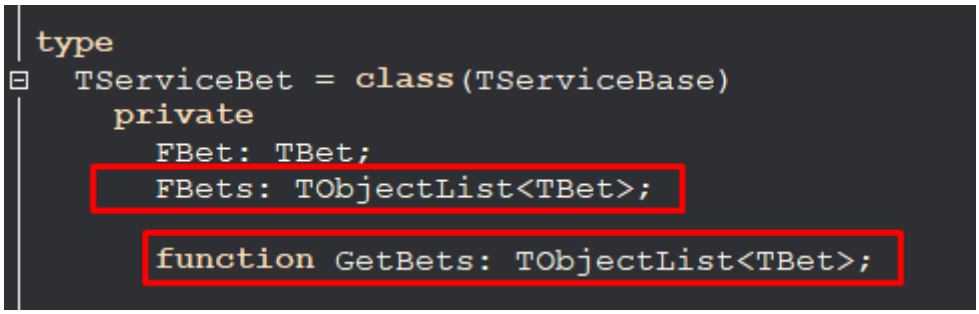


Figura 4 - UService.Bet

2.4. Clean Code

Além de seguir o conceito de *Baixo Acoplamento e Alta Coesão*, a aplicação possui alguns códigos limpos, como por exemplo evitando *Números Mágicos.*

```
procedure TfrmHome.lstMenuItemClick(const Sender: TCustomListBox;
  const Item: TListBoxItem);
begin
  Self.RemoverTelaAnterior;

  case TEnumMenu(Item.Index) of
    mnuTime:
      Self.AbrirTeam;
    mnuPartidas:
      Self.AbrirMatch;
    mnuPalpites:
      Self.AbrirBet;
    mnuSair:
      Self.Close;
  end;

  MultiView1.HideMaster;
end;
```

Figura 5 - UfrmHome

2.5. Herança

Foram utilizadas Heranças na aplicação a fim de evitar redundância de código entre as classes que possuem recursos semelhantes. Dessa forma trabalhamos com o conceito de classe Base, onde tudo que seja comum entre as classes filhas ficam nessa classe.

```
type
  TServiceBase = class(TInterfacedObject, IService)
    private
      FToken: String;
    protected
      FRESTClient: TRESTClient;
      FRESTRequest: TRESTRequest;
      FRESTResponse: TRESTResponse;

      procedure CarregarToken;

      procedure Registrar; virtual; abstract;
      procedure Listar; virtual; abstract;
      procedure Excluir; virtual; abstract;

      function ObterRegistro(const aId: Integer): TObject; virtual; abstract;
    public
      constructor Create;
      destructor  Destroy; override;
  end;
```

Figura 6 - UService.Base

```
type
  TServiceMatch = class(TServiceBase)
    private
      FMatch: TMatch;
      FMatchs: TObjectList<TMatch>;

      function GetMatchs: TObjectList<TMatch>;

      procedure PreencherMatchs(const aJsonMatchs: String);
    public
      constructor Create; overload;
      constructor Create(aMatch: TMatch); overload;
      destructor  Destroy; override;

      procedure Registrar; override;
      procedure Listar; override;
      procedure Excluir; override;

      function ObterRegistro(const aId: Integer): TObject; override;

      property Matchs: TObjectList<TMatch> read GetMatchs;
  end;
```

Figura 7 - UService.Match

2.6. Sobrecarga

Com a finalidade de facilitar o uso das classes utilizamos sobrecargas de métodos principalmente nos construtores. Dessa forma evitamos "*Ifs*" desnecessários em nossa aplicação. Sem contar que o código fica mais coeso.

```
public
  constructor Create; overload;
  constructor Create(const aId: Integer); overload;
  constructor Create(const aName: String); overload;
  constructor Create(const aId: Integer; aName: String); overload;
```

Figura 8 - UEntity.Teams

2.7. Sobrescrita e Polimorfismo

Como utilizamos o conceito de Classes Bases, foi necessário criarmos sobrescritas dos métodos declarados na classe base. Dessa forma cada classe filha executa o que for de específico dela dessa forma fazendo o uso dos mecanismos do polimorfismo.

```
procedure Registrar; virtual; abstract;
procedure Listar; virtual; abstract;
procedure Excluir; virtual; abstract;
```

Figura 9 - UService.Base

```
procedure Registrar; override;
procedure Listar; override;
procedure Excluir; override;
```

Figura 10 - UService.Bet

2.8. Interfaces

Para termos um código mais organizado e contribuir à uma boa evolução. Utilizamos interfaces para as classes bases implementarem. Dessa forma caso algum programador venha a "quebrar" o contrato da interface o próprio compilador irá "berrar".

```
type
  IService = interface
    procedure CarregarToken;
    procedure Registrar;
    procedure Listar;
    procedure Excluir;

    function ObterRegistro(const aId: Integer): TObject;
  end;
```

Figura 41 - UService.Intf

```
type
  TServiceBase = class(TInterfacedObject, IService)
    private
      FToken: String;
    protected
      FRESTClient: TRESTClient;
      FRESTRequest: TRESTRequest;
      FRESTResponse: TRESTResponse;

      procedure CarregarToken;

      procedure Registrar; virtual; abstract;
      procedure Listar; virtual; abstract;
      procedure Excluir; virtual; abstract;

      function ObterRegistro(const aId: Integer): TObject; virtual; abstract;
    public
      constructor Create;
      destructor  Destroy; override;
  end;
```

Figura 12 - UService.Base

2.9. DAO (Data Access Object)

Utilizamos o conceito de *Data Access Object (DAO).* Camada responsável por acessar objetos do banco de dados.

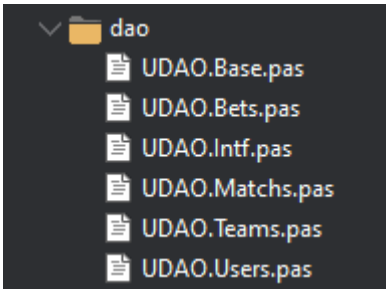


Figura 13 - model/dao

2.10. DTO (Data Transfer Object)

Utilizamos o conceito de *Data Transfer Object*. São classes dos nossos domínios de assuntos que utilizamos para "transportar" as informações entre as views e os services.

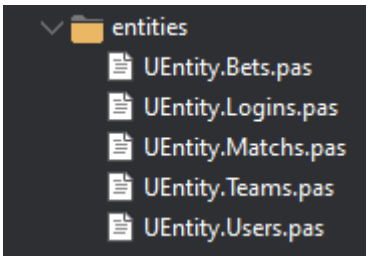


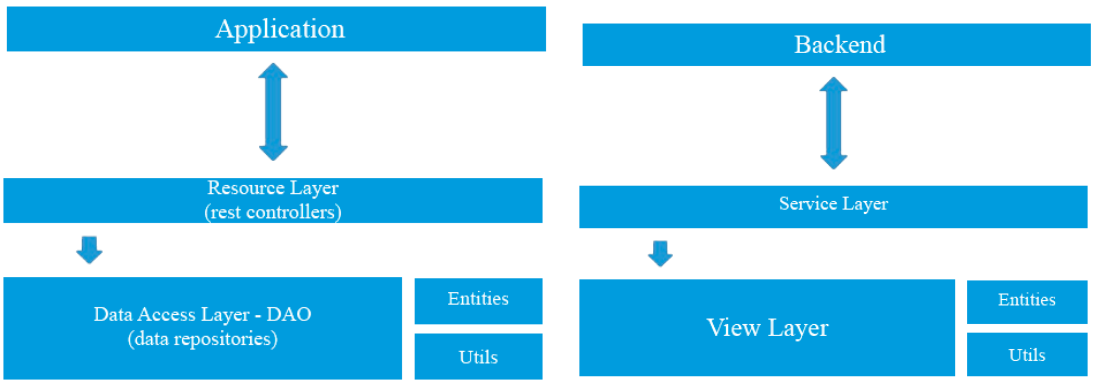
Figura 14 - model/entities

```
procedure TfraTeamRegistry.Registrar;
var
  xServiceTeam: IService;
begin
  if Trim(edtNome.Text) = EmptyStr then
    raise Exception.Create('Informe o Nome do Time.');

  xServiceTeam := TServiceTeam.Create(
    TTeam.Create(Trim(edtNome.Text)));

  xServiceTeam.Registrar;
  Self.VoltarTela;
end;
```

Figura 15 - UFraTeam.Registry

2.11. MVC (Model – View - Controller)

Foi utilizado uma espécie de padrão MVC no projeto.



2.12. Passagem por Referência

A fim de reduzir acoplamento entre as classes e com o intuito de digitarmos menos código, utilizamos a passagem por referência em alguns métodos.

```
procedure TServiceBet.CarregarUser(const aJsonUser: String; var aUser: TUser);
var
  xMemTable: TFDMemTable;
  xStatus: Byte;
begin
  aUser     := nil;
  xMemTable := TFDMemTable.Create(nil);

  try
    xMemTable.LoadFromJSON(aJsonUser);

    if xMemTable.RecordCount > 0 then
    begin
      xStatus := TUtilsFunctions.IIF<Byte>(
        xMemTable.FieldByName('status').AsString = 'true',
        1, 0);

      aUser := TUser.Create(
        xMemTable.FieldByName('id').AsInteger,
        xMemTable.FieldByName('name').AsString,
        xMemTable.FieldByName('login').AsString,
        xMemTable.FieldByName('password').AsString,
        xStatus);
    end;
  finally
    FreeAndNil(xMemTable);
  end;
```

Figura 15 - UService.Bet

```
procedure TServiceBet.PreencherBets(const aJsonBets: String);
var
  xMemTable: TFDMemTable;
  xMatch: TMatch;
  xUser: TUser;
  xStatus: Byte;
begin
  FBets.Clear;

  xMemTable := TFDMemTable.Create(nil);

  try
    xMemTable.LoadFromJSON(FRESTResponse.Content);

    while not xMemTable.Eof do
    begin
      Self.CarregarMatch(xMemTable.FieldByName('match').AsString,
                         xMatch);

      Self.CarregarUser(xMemTable.FieldByName('user').AsString,
                        xUser);

      xStatus := 0;
      if xMemTable.FieldByName('status').AsString = 'true' then
        xStatus := 1;

      FBets.Add(TBet.Create(
        xMemTable.FieldByName('id').AsInteger,
        xMatch,
        xMemTable.FieldByName('result_Team_A').AsInteger,
        xMemTable.FieldByName('result_Team_B').AsInteger,
        xStatus,
        xUser));
```

Figura 16 - UService.Bet