PPW (*Pascal Pages for Web*)
- Manual do Programador -

Versão 1.0

Por João Werther Filho

ÍNDICE

Índice 2
Introdução 3
Visão Geral 4
Componentes 5
TStringsList 6
TAttributesList 7
TElementsList 8
TXMLFile 10
TRequestParameter 12
TClientSession 13
TConnectionVariable 15
TCookie 16
TSessionVariable 17
Desenvolvimento e Distribuição (deployment) 20
Configuração do Arquivo APW.XML 23
O Processo de Distribuição PPW 25
Servidor PPW 28
Iniciando Servidor PPW 30
Requisições ao Servidor PPW 31
Tratamento de Cookies 32

INTRODUÇÃO

Este documento possui a finalidade de orientar o programador PPW na utilização completa e suficiente dos recursos desta tecnologia, bem como trazer um pouco mais de entendimento técnico do seu funcionamento.

Sua utilização ou leitura é recomendada por todos que desejarem ingressar no desenvolvimento de aplicações PPW

VISÃO GERAL

A tecnologia PPW conjuga um novo ambiente de desenvolvimento de aplicações web baseado na linguagem Pascal, utilizando páginas HTML dinâmicas, com execução baseada em tecnologia *server-side* de processamento CGI. Esta nova tecnologia permite que o programador desenvolva suas páginas web em qualquer ferramenta de página dinâmica da sua preferência, utilizando código Pascal incluso nas páginas HTML através de delimitadores do código. A idéia é que o PPW forneça a mesma praticidade de desenvolvimento de outras tecnologias que utilizam páginas dinâmicas, como o ASP, PHP ou JSP, associada à rapidez dos executáveis de aplicações CGI.

A linguagem adotada foi o *Object Pascal*, que é basicamente a linguagem Pascal com extensão de orientação a objeto, tornando-a ainda mais poderosa. Para compilador da linguagem, foi selecionado o compilador da **Borland Corp.**, utilizado no *Delphi 7*. Sua opção é fortemente justificada porque as classes do *Delphi 7* são chamadas de *Cross-Platforms*, ou seja, podem ser compiladas e executadas tanto em ambiente Windows como Linux. Apesar dessas facilidades, foi decidido que serão utilizadas apenas poucas classes oriundas do *Delphi 7*, visando uma maior compatibilidade futura. A maioria das funções e classes necessárias à tecnologia foi construída. Basicamente, do *Delphi 7*, só foi utilizado:

SysUtils: a *unit* de utilitários gerais; SQLExpr: a *unit* de utilitários de acesso à banco de dados, que dá suporte ao *dbExpress* do *Delphi 7* (componente de interface com banco de dados); IdTCPServer e a IdTCPClient: as *units* que simulam servidores e clientes TCP.

Todo acesso a banco de dados é feita pelo *dbExpress*; consequentemente os bancos que podem ser acessados são os disponibilizado pelo *dbExpress* do *Delphi 7*;

Como servidor HTTP é utilizado o já bastante conhecido servidor HTTP Apache, v2.0.48, da **Apache Software Foundation**.

Em linhas gerais, a tecnologia PPW permite ao desenvolvedor:

- Utilizar páginas dinâmicas PPW e páginas estáticas HTML;
- Utilizar qualquer ambiente de desenvolvimento para páginas HTML ou dinâmicas, como *Dreamweaver*, *HomeSite* e outros;
- Receber dados e variáveis através de POST e GET;
- Controlar, validar e expirar sessões HTTP;
- Utilizar variáveis de sessão;
- Acessar diretamente bancos de dados *MySQL, MS-SQL Server, Oracle, IBM DB2, Interbase, Informix* (de acordo com disponibilizado pelo *dbExpress* do *Delphi 7*);
- Utilizar conexões inteligentes de banco de dados, estabelecendo apenas uma única vez durante todo o processamento da sessão;
- Definir e remover *cookies*;
- Utilizar *units* Pascal referenciadas pelas páginas PPW, facilitando a separação entre interface e código – O código não precisa estar todo na página;
- Obter alto desempenho das aplicações, devido ao processamento CGI.

COMPONENTES

A tecnologia PPW é composta por um ambiente de desenvolvimento e distribuição (*deployment*) das suas aplicações, baseado em páginas dinâmicas (páginas PPW); e outro que cuida exclusivamente da tecnologia necessária à sua execução, baseado em processamento server-side CGI. Ela contém dois aplicativos básicos, somados a um conjunto de classes e componentes.

Os dois aplicativos, de vital importância à tecnologia, são o *deployPPW* e o *PPWController*. Ambos são programas executáveis e foram desenvolvidos em *Object Pascal* com o *Delphi 7*, utilizando um conjunto de classes que será visto a seguir.

Existem componentes e classes que suportam estes dois aplicativos, além de oferecer funcionalidades extras à tecnologia. Muitas dessas classes e componentes são totalmente transparentes ao programador.

Tabela 1. Classes utilizadas pelo PPW

Nome da Classe	**Descrição**
TRequestParameter	Trata parâmetros recebidos por POST ou GET
TClientSession	Traduz sessões estabelecidas da aplicação com máquinas clientes
TSessionVariable:	Trata variáveis de sessão
TConnectionVariable:	Corresponde a variáveis de conexão a banco, vinculadas à sessões de cliente
TCookie	Manipula os *cookies* de uma aplicação
TXMLFile	Trata de arquivos XML
TElementsList	Representa elementos (tags) de um arquivo XML
TAttributesList	Representa os atributos de elementos (tags) de um arquivo XML
TStringsList	Representa lista encadeada/dinâmica de strings
SqlExpr	*Unit* Pascal que oferece um conjunto de classes que implementam o *dbExpress* para acesso a banco de dados, disponibilizada pelo *Delphi 7*
IdTCPServer e IdTCPClient	*Units* Pascal que oferecem um conjunto de classes que implementam respectivamente um servidor TCP *multi-thread* e um cliente TCP, ambas disponibilizada pelo *Delphi 7*
TControllerServer	Classe herdeira de *TIdTCPServer*, especializando o tratamento do servidor TCP para o objeto *Servidor*, instância desta classe, que é o coração do *PPWController*

Será apresentada a estrutura das classes desenvolvidas diretamente para a tecnologia, em ordem inversa de complexidade. Como as classes provenientes das *units SqlExpr*, *IdTCPServer* e *IdTCPClient* são oriundas do *Delphi 7*, não serão abordadas nesta seção.

TStringsList

O objetivo desta classe é gerar objetos que simulem o comportamento de listas de strings simplesmente encadeadas por ponteiros. Cada nó (ou elemento) da lista é composto por um item (string) e um ponteiro apontando para o próximo nó. Como é lista encadeada por ponteiros, seu tamanho é restrito apenas pela memória disponibilizada pelo SO.

Os tipos construídos específicos para esta classe estão descritos a seguir, referenciados inclusive pela cor cinza.

```
TPString = ^TNodeString;
TNodeString = record
                Item: string;
                Next: TPString;
              end;
```

TNodeString representa cada nó (elemento) da lista e *TPString* um ponteiro para este nó.

Os métodos disponibilizados por esta classe são os seguintes:

Create: Método construtor da classe. Onde a lista é inicializada.

Add(string): Adiciona a string no final da lista. Retorna verdadeiro se operação bem sucedida.

Insert(string; TPString): Insere a string na posição apontada por *TPString*. Retorna verdadeiro se operação bem sucedida.

Insert(string; integer): Insere a string na posição fornecida pelo parâmetro *integer*. Retorna verdadeiro se operação bem sucedida.

Find(string): Busca o primeiro elemento da lista que corresponda ao parâmetro string. Retorna o ponteiro *TPString* associado ao elemento.

Remove(TPString): Remove o próximo elemento referenciado pelo parâmetro *TPString*. Retorna a string do elemento removido.

Remove(integer): Remove o elemento que está na posição fornecida pelo parâmetro *integer*. Retorna a string do elemento removido.

IsEmpty: Verifica se a lista está vazia. Retorna verdadeiro caso esteja.

Show: Exibe todas as strings elementos da lista, separados por espaço em branco.

WriteTxt(string): Cria um arquivo texto baseado nos elementos da lista, de nome e local especificado pelo parâmetro string. Preenche cada linha do arquivo para cada elemento string da lista. Retorna verdadeiro se operação bem sucedida.

GetItem(integer): Retorna a string que está na posição da lista indicado pelo parâmetro *integer*.

A seguir, a definição da classe no código *Object Pascal*, junto com os seus tipos construídos de variáveis, estes últimos especificados na cor cinza:

```
TStringsList = class (TObject)
  private
    First: TPString;
    Last: TPString;
```

```
    Sz: integer;
  protected
    function Remove(pAnt: TPString): string; overload;
    function GetItem(pos:integer): string;
  public
    constructor Create;
    function IsEmpty: boolean;
    function Add(Item:string): boolean;
    function Insert(Item:string; pos:integer): boolean; overload;
    function Insert(Item:string; p:TPString): boolean; overload;
    function Remove(pos: integer): string; overload;
    function Find(Item:string): TPString;
    function WriteTxt(sNomeArq:string): boolean;
    procedure Show;
    property Item[Indice: integer]: string read GetItem;
    property Size: integer read Sz;
  end;
```

TAttributesList

O objetivo desta classe é gerar objetos que representem uma lista de atributos específicos para um determinado elemento (tag) XML. Cada nó (ou elemento) da lista corresponde a um atributo que é composto por nome e valor do atributo (*Attrib*) e um ponteiro apontando para o próximo nó.

Os tipos construídos específicos para esta classe estão descritos a seguir, referenciados inclusive pela cor cinza.

```
TPAttrib = ^TNodeAttrib;
TAttrib = record
            Name: string;
            Value: string;
          end;
TNodeAttrib = record
                 Attrib: TAttrib;
                 Next: TPAttrib;
              end;
```

TNodeAttrib representa cada nó (elemento) da lista e *TPAttrib* um ponteiro para este nó. *TAttrib* representa as informações do atributo, nome e valor.

Por se tratar basicamente de uma lista encadeada, os métodos disponibilizados por esta classe, listados a seguir, são muito semelhantes à classe *TStringsList* vista anteriormente:

Create: Método construtor da classe. Onde a lista é inicializada.

Add(TAttrib): Adiciona um novo atributo – nome e valor – ao final da lista de atributos de um elemento (tag). Retorna verdadeiro se operação bem sucedida.

Insert(TAttrib; TPAttrib): Insere um novo atributo – nome e valor – na posição apontada por *TPAttrib*. Retorna verdadeiro se operação bem sucedida.

Insert(TAttrib; integer): Insere um novo atributo – nome e valor – na posição fornecida pelo parâmetro *integer*. Retorna verdadeiro se operação bem sucedida.

Find(string): Busca o primeiro atributo da lista cujo nome corresponde ao parâmetro string. Retorna o ponteiro *TPAttrib* associado ao atributo.

Remove(TPAttrib): Remove o próximo atributo referenciado pelo parâmetro *TPAttrib*. Retorna o atributo (*TAttrib*) removido.

Remove(integer): Remove o atributo que está na posição fornecida pelo parâmetro *integer*, dentro do elemento (tag) referenciado. Retorna o atributo (*TAttrib*) removido.

IsEmpty: Verifica se a lista de atributos do elemento (tag) está vazia. Retorna verdadeiro caso esteja.

GetValue(string): Retorna o valor do atributo do elemento (tag), cujo nome corresponde ao parâmetro string fornecido.

A seguir, a definição da classe no código *Object Pascal*, junto com os seus tipos construídos de variáveis, estes últimos especificados na cor cinza:

```
TAttributesList = class (TObject)
  private
    First: TPAttrib;
    Last: TPAttrib;
    Sz: integer;
  protected
    constructor Create;
    function Add(Attrib:TAttrib): boolean;
    function Insert(Attrib:TAttrib; pos:integer): boolean; overload;
    function Insert(Attrib:TAttrib; p:TPAttrib): boolean; overload;
    function Remove(pAnt: TPAttrib): TAttrib; overload;
    function Remove(pos: integer): TAttrib; overload;
    function Find(AttribName:string): TPAttrib;
    function GetValue(AttribName:string): string;
    function IsEmpty: boolean;
    property Size: integer read Sz;
    property Value[AttribName:string]: string read GetValue;
  end;
```

Note que esta classe não possui nenhum método público, o que significa que ela apenas é de uso interno da tecnologia.

TElementsList

O objetivo desta classe é gerar objetos que representem uma lista de elementos (tags) de todo um arquivo XML. Cada nó da lista corresponde a uma tag XML que é composto pelo tipo construído *TElement*, que possui nome e valor da tag, nível de profundidade da tag dentro do arquivo, lista de atributos da tag e se está completa (se existe abertura e fechamento da tag). Composto também por dois ponteiros, um apontando para o próximo nó e outro apontando para o nó anterior, simulando assim uma lista duplamente encadeada.

Os tipos construídos específicos para esta classe estão descritos a seguir, referenciados inclusive pela cor cinza.

```
TPElement = ^TNodeElement;
TElement = record
             TagName: string;
```

```
              Value: string;
              Level: integer;
              Attributes: TAttributesList;
              Complete: boolean;
            end;
TNodeElement =  record
                  Element: TElement;
                  Next: TPElement;
                  Previous: TPElement;
                end;
```

TNodeElement representa cada nó (elemento) da lista e *TPElement* um ponteiro para este nó. *TElement*, como foi visto, representa as informações pertinentes de uma tag XML.

Os métodos disponibilizados por esta classe são os seguintes:

Create: Método construtor da classe. Onde a lista de tags XML é inicializada.

CreateElement(TElement): Método de classe que inicializa os dados necessários a um elemento (tag), inclusive a sua lista de atributos associada.

Add(TElement): Adiciona um novo elemento (tag) ao final da lista. Retorna verdadeiro se operação bem sucedida.

Insert(TElement; TPElement): Insere um novo elemento (tag) na posição apontada por *TPElement*. Retorna verdadeiro se operação bem sucedida.

Insert(TElement; integer): Insere um novo elemento (tag) na posição fornecida pelo parâmetro *integer*. Retorna verdadeiro se operação bem sucedida.

Find(string): Busca o primeiro elemento (tag) da lista cujo nome corresponde ao parâmetro string. Retorna o ponteiro *TPElement* associado ao elemento.

Remove(TPElement): Remove o próximo atributo referenciado pelo parâmetro *TPElement*. Retorna o elemento (*TElement*) removido.

Remove(integer): Remove o elemento que está na posição fornecida pelo parâmetro *integer*. Retorna o elemento (*TElement*) removido.

IsEmpty: Verifica se a lista de elementos está vazia. Retorna verdadeiro caso esteja.

GetValue(string): Retorna o valor do primeiro elemento (tag) da lista, cujo nome corresponde ao parâmetro string fornecido.

ValidXML: Verifica se o arquivo XML carregado para esta lista de elementos possui uma estrutura válida. Considera-se estrutura válida quando todas as tags que foram abertas também foram fechadas. Retorna verdadeiro caso a estrutura seja válida.

Clear: Limpa a lista, liberando todos os ponteiros e memória alocada da lista.

Show: Exibe todos os elementos (tags) da lista, cada um por linha, com os respectivos atributos.

A seguir, a definição da classe no código *Object Pascal*, junto com os seus tipos construídos de variáveis, estes últimos especificados na cor cinza:

```
TElementsList = class (TObject)
  private
```

```
    First: TPElement;
    Last: TPElement;
    Sz: integer;
    Level: integer;
  protected
    constructor Create;
    class procedure CreateElement(var element:TElement);
    procedure Clear;
    procedure Show;
    function Add(Element:TElement): boolean;
    function Insert(Element:TElement; p:TPElement): boolean; overload;
    function Insert(Element:TElement; pos:integer): boolean; overload;
    function Remove(pAnt: TPElement): TElement; overload;
    function Remove(pos: integer): TElement; overload;
    function Find(TagName:string): TPElement;
    function GetValue(TagName:string): string;
    function ValidXML: boolean;
    function IsEmpty: boolean;
    property Size: integer read Sz;
    property Value[TagName:string]: string read GetValue;
    property CurrentLevel: integer read Level write Level;
  end;
```

Note que esta classe não possui nenhum método público, o que significa que ela apenas é de uso interno da tecnologia.

TXMLFile

O objetivo desta classe é gerar objetos que representem todo um arquivo XML e permitir que se navegue entre todas as suas tags, semelhantes às classes DOM (*Document Objet Model*) da plataforma java. A classe possui três atributos básicos: o nome do arquivo XML, uma lista de elementos (tags) do arquivo XML – do tipo *TElementsList* – e um ponteiro *TPElement* que aponta para a tag corrente de navegação do arquivo.

As tags dos arquivos XML são carregados no atributo TagList – do tipo TElementsList – e esta classe *TXMLFile* basicamente oferece métodos para percorrer todo o arquivo pelas suas tags.

Os métodos disponibilizados por esta classe são os seguintes:

Create(TFileName): Método construtor da classe. Onde é carregado todo o arquivo XML no atributo TagList. Este método, na verdade, apenas lê as linhas dos arquivos e coloca todo o arquivo em uma grande string, para em seguida chamar o método *LoadElements*.

LoadElements(string): Recebe a string gerada no método construtor, carrega toda a TagList – atributo que é um objeto de *TElementsList* – e verifica se possui estrutura XML válida. Utiliza o método *UpdateTag* como auxiliar de identificação e criação de tags. Retorna verdadeiro se operação bem sucedida.

UpdateTag(string,string): Atualiza a tag identificada no primeiro parâmetro string com o valor do segundo parâmetro string. Cria a tag no objeto TagList se ainda não existir. Retorna verdadeiro se operação bem sucedida.

GetCurrentTagName e *GetValue*: Retornam, respectivamente, o nome da tag e o valor da tag corrente da navegação.

GetAttributeValue(string): Retorna o valor do atributo especificado pelo parâmetro string, atributo este pertencente à tag corrente da navegação.

FindNodeByTagName(string): Localiza a primeira tag, cujo nome corresponda ao parâmetro string. Caso exista, sinaliza ela como a tag corrente de navegação, através do seu atributo PCurrentTag que é um ponteiro de *TElement*. Retorna verdadeiro se operação bem sucedida.

FirstNode: Posiciona a primeira tag como sendo a tag corrente de navegação. Retorna verdadeiro se operação bem sucedida.

FirstChild: Posiciona na primeira tag filha da tag corrente de navegação. Tag filhas são as tags mais internas, situadas entre a abertura e fechamento da tag em questão. Retorna verdadeiro se operação bem sucedida.

LastChild: Posiciona na última tag filha da tag corrente de navegação. Retorna verdadeiro se operação bem sucedida.

NextSibling: Posiciona na próxima tag irmã da tag corrente de navegação. Tag irmã são tags que estão no mesmo nível em um XML, não sendo pai nem filha da tag em questão. A próxima tag irmã é a tag que a sucede. Retorna verdadeiro se operação bem sucedida.

PreviousSibling: Posiciona na tag irmã anterior da tag corrente de navegação. A tag irmã anterior é a tag que a precede. Retorna verdadeiro se operação bem sucedida.

ParentNode: Posiciona na tag pai da tag corrente de navegação. Uma tag é pai de outra tag quando esta última é filha dela, conforme explicado antes. Retorna verdadeiro se operação bem sucedida.

A seguir, a definição da classe no código *Object Pascal*:

```
TXMLFile = class (TObject)
  private
    FileName: TFileName;
    TagList : TElementsList;
    PCurrentTag: TPElement;
    IsEmpty: boolean;
  protected
    function GetValue: string;
    function GetAttributeValue(AttrName:string): string;
    function GetCurrentTagName: string;
    function LoadElements(Content: string): boolean;
    function UpdateTag(TagDesc,TagValue:string): boolean;
  public
    constructor Create(XMLFileName:TFileName);
    function FindNodeByTagName(TagName:string): boolean;
    function FirstNode: boolean;
    function FirstChild: boolean;
    function LastChild: boolean;
    function NextSibling: boolean;
    function PreviousSibling: boolean;
    function ParentNode: boolean;
    property TagValue: string read GetValue;
```

```
      property AttributeValue[AttrName:string]:string read
GetAttributeValue;
      property CurrentTagName: string read GetCurrentTagName;
    end;
```

TRequestParameter

O objetivo desta classe é gerar objetos que representem listas de parâmetros recebidos de uma página HTML através de métodos POST ou GET. O PPW já instancia e disponibiliza ao programador um objeto (*Parameter*) desta classe, como será mostrado mais adiante.

Cada nó (ou elemento) da lista de parâmetros mencionada é composto pelo parâmetro e um ponteiro apontando para o próximo nó. O parâmetro é composto por seu nome e uma coleção (ou lista) de valores – A necessidade de uma coleção se justifica no fato de que podem ocorrer vários parâmetros com o mesmo nome e valores diferentes em uma mesma tela submetida. Como é lista encadeada por ponteiros, seu tamanho é restrito apenas pela memória disponibilizada pelo SO.

Os tipos construídos específicos para esta classe estão descritos a seguir, referenciados inclusive pela cor cinza.

```
  TPParam = ^TNodeParam;
  TItemParam = record
                 Id: string;
                 Value: TStringsList;
               end;
  TNodeParam = record
                 Item: TItemParam;
                 Next: TPParam;
               end;
```

TNodeParam representa cada nó (elemento) da lista e *TPParam* um ponteiro para este nó. *TItemParam* é o tipo construído utilizado para ao parâmetro.

Os métodos disponibilizados por esta classe são os seguintes:

Create(string): Método construtor da classe. Onde através do parâmetro string é fornecida a lista de parâmetros provenientes de métodos de submissão de página POST e GET. A lista é inicializada e carregada com esses valores. Este parâmetro string possui dados dispostos de forma semelhante a: "?<nome_param1> = <valor1>&<nome_param2> = <valor2>". Se existirem vários parâmetros com o mesmo nome são multivalorados em uma lista de strings (*TStringsLists*) vinculados a um único Id.

Add(TItemParam): Adiciona um parâmetro ao final da lista. Retorna verdadeiro se operação bem sucedida.

Insert(TItemParam; integer): Insere o parâmetro na posição fornecida pelo parâmetro *integer*. Retorna verdadeiro se operação bem sucedida.

Find(string): Busca o parâmetro da lista cujo nome corresponde a string fornecida. Retorna o ponteiro *TPParam* associado ao parâmetro localizado.

Remove(TPParam): Remove o próximo parâmetro referenciado por *TPParam*. Retorna o parâmetro (nome e coleção de valores) removido.

Remove(integer): Remove o parâmetro que está na posição fornecida por *integer*. Retorna o parâmetro (nome e coleção de valores) removido.

IsEmpty: Verifica se a lista está vazia. Retorna verdadeiro caso esteja.

GetValues(string): Retorna a coleção (ou lista) de valores associados ao parâmetro cujo nome (ou Id) é fornecido por string.

GetValue(string,integer): Retorna um único valor associado ao parâmetro cujo nome (ou Id) é fornecido por string. O *integer* determina a posição do valor desejado na coleção de valores.

A seguir, a definição da classe no código *Object Pascal*, junto com os seus tipos construídos de variáveis, estes últimos especificados na cor cinza:

```
TRequestParameter = class (TObject)
  private
    First: TPParam;
    Last: TPParam;
    Sz: integer;
  protected
    function Add(Item:TItemParam): boolean;
    function Insert(Item:TItemParam;pos:integer): boolean;
    function Remove(pAnt: TPParam): TItemParam; overload;
    function Remove(pos: integer): TItemParam; overload;
    function Find(Id:string): TPParam;
    function GetValues(Id:string): TStringsList;
    function GetValue(Id:string;pos:integer): string;
  public
    constructor Create(HTTPRequest:string);
    function IsEmpty: boolean;
    property Size: integer read Sz;
    property Values[Id:string]: TStringsList read GetValues;
    property Value[Id:string]: string index 1 read GetValue;
  end;
```

TClientSession

O objetivo desta classe é gerar objetos que representem sessões web estabelecidas pelo cliente HTTP no servidor PPW. Cada objeto representa uma sessão única do usuário e ele tem a finalidade de oferecer ao usuário as funcionalidades de sessão disponíveis na tecnologia PPW. Estas funcionalidades, como atuam na camada cliente, fazem a interface com a camada do servidor PPW, pois as informações de sessão são guardadas nesta última camada. O PPW já instancia e disponibiliza ao programador um objeto (*Session*) desta classe, como será mostrado mais adiante.

Os métodos disponibilizados por esta classe são os seguintes:

Create: Método construtor da classe. Para estabelecer conexões com o *PPWController*, é necessário a instanciação de um objeto de *TIdTCPClient*, que tem exatamente esta finalidade. Mas, para isso, é necessário que a própria aplicação que o instanciou, e instanciou também o objeto de *TClientSession*, se torne a proprietária (*owner*) destes objetos. Este método tem a necessidade de guardar esta referencia como atributo do objeto.

Add(string1; string2): Cria uma variável de sessão de nome *string1* com o valor *string2*. Ele faz uma chamada ao *PPWController* para isso. Mais detalhes sobre utilização de variáveis de sessão em Requisições ao Servidor PPW.

Add(string; TSQLConnection): Cria uma variável de sessão correspondendo a uma conexão que estabelece e retorna através do parâmetro *TSQLConnection*. Os atributos de conexão estão no arquivo *apw.xml*, como será visto mais adiante e será detalhado ainda mais em Configuração do Arquivo APW.XML. O parâmetro string identifica a conexão em *apw.xml* e também no lado do servidor PPW. Para realizar este método, ele faz uma chamada ao *PPWController*, que tenta estabelecer a conexão. Mais detalhes sobre estabelecimento e obtenção de conexão a banco em Requisições ao Servidor PPW.

GetConnection(string): Obtém uma conexão de nome do parâmetro string já estabelecida, ou estabelece-a de acordo com os parâmetros da conexão de nome string constante em *apw.xml*. Retorna uma variável *TSQLConnection* relativo a esta conexão. Para realizar este método, ele faz uma chamada ao *PPWController*.

GetValue(string): Retorna o valor da variável de sessão de nome fornecido pelo parâmetro string. Para isso, ele faz uma chamada ao *PPWController*.

Update(string1; string2): Atualiza uma variável de sessão de nome *string1* com o valor *string2*. Ele faz uma chamada ao *PPWController* para isso.

Remove(string): Remove uma variável de sessão de nome *string1*. Para isso, faz uma chamada ao *PPWController*.

ValidSession: Retorna verdadeiro se a sessão está válida e ainda não expirou por tempo sem acesso.

SetTimeToExpire(*double*): Atribui um tempo de expiração por acesso (tempo em que a sessão ficou sem requisição ao servidor). O parâmetro *double* se refere ao tempo, expresso em segundos.

Invalidate: Invalida a sessão, expirando-a imediatamente.

A seguir, a definição da classe no código *Object Pascal*:

```
TClientSession = class (TObject)
  private
    AOwner: TComponent;
  protected
  public
    constructor Create(AOwner:TComponent);
    procedure Add(Name,Value:string); overload;
    procedure Add(Name:string; conn:TSQLConnection); overload;
    procedure Remove(Name:string);
    procedure Update(Name,Value:string);
    function GetConnection(Name:string): TSQLConnection;
    function GetValue(Name:string): string;
    procedure SetTimeToExpire(TimeToExpire: Double);
    function ValidSession: boolean;
    procedure Invalidate;
  end;
```

TConnectionVariable

O objetivo desta classe é gerar objetos que simulem o comportamento de listas de objetos de conexão a banco de dados, simplesmente encadeadas por ponteiros. Cada nó (ou elemento) da lista é composto por um item (o objeto de conexão) e um ponteiro apontando para o próximo nó. Como é lista encadeada por ponteiros, seu tamanho é restrito apenas pela memória disponibilizada pelo SO.

Os tipos construídos específicos para esta classe estão descritos a seguir, referenciados inclusive pela cor cinza.

```
TPCVar = ^TNodeCVar;
TItemCVar = record
              conn: TSQLConnection;
            end;
TNodeCVar = record
              Item: TItemCVar;
              Next: TPCVar;
            end;
```

TNodeCVar representa cada nó (elemento) da lista e *TPCvar* um ponteiro para este nó. Note que *TItemCVar*, que representa o item de cada nó, é constituído apenas por um objeto *TSQLConnection*, que é uma classe já disponível na *unit SQLExpr* do Delphi 7. A finalidade do *TSQLConnection* é gerar objetos de conexão a banco de dados.

Os métodos disponibilizados por esta classe são os seguintes:

Create: Método construtor da classe. Onde a lista é inicializada.

Add(TItemCVar): Adiciona o item de conexão *TItemCVar* ao final da lista. Retorna verdadeiro se operação bem sucedida.

Find(string): Busca o primeiro item da lista, cujo nome da conexão corresponda ao parâmetro string. Retorna o item *TItemCVar* relativo à conexão solicitada.

FindPos(string; TPCVar): Busca o primeiro elemento da lista, cujo nome da conexão corresponda ao parâmetro string. Retorna o item *TPCVar* relativo à conexão solicitada. O parâmetro *TPCVar*, como é passado por referência, retorna o ponteiro do item anterior para facilitar operações na lista.

Remove(TPCVar): Remove o próximo elemento referenciado pelo parâmetro *TPCVar*. Retorna o item *TItemCVar* referente ao elemento removido.

Remove(string): Remove o elemento cujo item possui nome de conexão correspondente ao parâmetro string. Retorna verdadeiro se operação bem sucedida.

IsEmpty: Verifica se a lista está vazia. Retorna verdadeiro caso esteja.

A seguir, a definição da classe no código *Object Pascal*, junto com os seus tipos construídos de variáveis, estes últimos especificados na cor cinza:

```
TConnectionVariable = class (TObject)
  private
    First: TPCVar;
    Last: TPCVar;
    Sz: integer;
```

```
protected
  function Remove(pAnt: TPCVar): TItemCVar; overload;
  function FindPos(ConnName:string; var pAnt:TPCVar): TPCVar;
public
  constructor Create;
  function Add(Item:TItemCVar): boolean;
  function Find(ConnName:string): TItemCVar;
  function Remove(ConnName:string): boolean; overload;
  procedure Clear;
  function IsEmpty: boolean;
  property Size: integer read Sz;
end;
```

TCookie

O objetivo desta classe é criar objeto para gerenciar os *cookies* referentes a uma aplicação em um cliente. O PPW já instancia e disponibiliza ao programador um objeto (*Cookie*) desta classe, como será mostrado mais adiante.

Os *cookies* daquela aplicação/cliente ficam armazenados na aplicação em forma de lista simplesmente encadeada por ponteiros. Cada nó (ou elemento) da lista é composto por um item (o *cookie*) e um ponteiro apontando para o próximo nó. Como é lista encadeada por ponteiros, seu tamanho é restrito apenas pela memória disponibilizada pelo SO.

Os tipos construídos específicos para esta classe estão descritos a seguir, referenciados inclusive pela cor cinza.

```
TPCookie = ^TNodeCookie;
TItemCookie = record
                Name: string;
                Value: string;
                Expires: string;
              end;
TNodeCookie = record
                Item: TItemCookie;
                Next: TPCookie;
              end;
```

TNodeCookie representa cada nó (elemento) da lista e *TPCookie* um ponteiro para este nó. Note que *TItemCookie*, que representa o item de cada nó, é constituído apenas por nome, valor e data de expiração do *cookie*.

Os métodos disponibilizados por esta classe são os seguintes:

Create: Método construtor da classe. Onde a lista é inicializada.

Create(string): Segundo método construtor da classe. Além da lista ser inicializada, ela é carregada com os *cookies* disponíveis na sessão, informados através do parâmetro string. Este parâmetro possui dados dispostos de forma semelhante a: "<nome_cookie1> = <valor1>; <nome_cookie2> = <valor2>". *Cookies* de nomes iguais são sobrepostos valendo apenas a definição do último.

Add(TItemCookie): Adiciona o item de *cookie TItemCookie* ao final da lista. Retorna verdadeiro se operação bem sucedida.

Add(string1; string2; string3): Solicita a criação de um *cookie* de nome *string1* e valor *string2*, com data de expiração indicada em *string3* na máquina cliente. A data de expiração deve ser informada no formato "Wdy, DD-Mon-YYYY HH:MM:SS GMT" (exemplos: "qua, 12-out-2005" ou "qui, 13-out-2005 09:30"). Data de expiração em branco ou fora deste formato significa que a duração do *cookie* está restrita à sessão atual. Este método adiciona também o item de *cookie TItemCookie* ao final da lista.

Find(string): Busca o primeiro item da lista de *cookies*, cujo nome corresponda ao parâmetro string. Retorna o item *TItemCookie* relativo ao *cookie* solicitado.

IsEmpty: Verifica se a lista está vazia. Retorna verdadeiro caso esteja.

GetValue(string): Retorna o valor do *cookie* de nome fornecido pelo parâmetro string..

A seguir, a definição da classe no código *Object Pascal*, junto com os seus tipos construídos de variáveis, estes últimos especificados na cor cinza:

```
TCookie = class (TObject)
  private
    First: TPCookie;
    Last: TPCookie;
    Sz: integer;
  protected
    function Add(Item:TItemCookie): boolean; overload;
    function Find(Name:string): TPCookie;
    function GetValue(Name:string): string;
  public
    constructor Create; overload;
    constructor Create(HTTPCookies:string); overload;
    function IsEmpty: boolean;
    procedure Add(Name,Value,Expires:string); overload;
    property Value[Id:string]: string read GetValue;
  end;
```

TSessionVariable

O objetivo desta classe é criar objetos correspondentes a uma sessão web estabelecida, contendo variáveis, conexões e informações necessárias a uma sessão. Nesta estrutura, as variáveis de sessão são armazenadas em listas simplesmente encadeadas por ponteiros. As conexões a banco referente à sessão são armazenadas em uma variável/objeto interno do tipo *TConnectionVariable*. Além destas informações, cada objeto desta classe contém o número de identificação da sessão no servidor PPW, contém também a aplicação correspondente, a data/hora da última requisição ao servidor desta sessão e a data/hora de expiração da mesma.

Os tipos construídos específicos para esta classe estão descritos a seguir, referenciados inclusive pela cor cinza.

```
TPSVar = ^TNodeSVar;
TItemSVar = record
              Name: string;
              Value: string;
            end;
TNodeSVar = record
```

```
    Item: TItemSVar;
    Next: TPSVar;
  end;
```

TNodeSVar representa cada nó (elemento) da lista de variáveis de sessão e *TPSVar* um ponteiro para este nó. Note que *TItemSVar*, que representa o item variável de sessão de cada nó, é constituído apenas por nome e valor.

Os métodos disponibilizados por esta classe são os seguintes:

Create: Método construtor da classe. Onde a lista de variáveis de sessão é inicializada, bem como todos os outros atributos de objeto. Neste método é estabelecida a expiração padrão de uma sessão web de servidor PPW como sendo de trinta minutos

SetAppId(string1; string2): Atribuído o nome da aplicação com o valor fornecido pelo parâmetro *string1* e o número identificador da sessão com o valor fornecido por *string2*.

SessionExpired: Testa se a sessão está expirada. Retorna verdadeiro caso esteja. Se estiver, também é executado o método *Clear*.

AddTimeToExpire(TDateTime): Adiciona um valor *TDateTime* à data/hora de expiração. Este valor é fornecido em unidade de dia.

SetTimeToExpire(TDateTime): Atribui um valor *TDateTime* para data/hora de expiração. Este valor é fornecido em unidade de dia.

Add(TItemSVar): Adiciona o item correspondente a uma nova variável de conexão *TItemCVar* ao final da lista de variáveis de sessão. Retorna verdadeiro se operação bem sucedida.

Find(string): Busca o primeiro item da lista de variáveis de sessão, cujo nome da variável corresponda ao parâmetro string. Retorna o item *TItemSVar* relativo à variável solicitada.

FindPos(string; TPSVar): Busca o primeiro elemento da lista de variáveis de sessão, cujo nome da conexão corresponda ao parâmetro string. Retorna o item *TPSVar* relativo à variável de sessão solicitada. O parâmetro *TPSVar*, como é passado por referência, retorna o ponteiro do item anterior para facilitar operações na lista.

Update(string1; string2): Atualiza a variável de sessão de nome fornecido pelo parâmetro *string1* com o valor fornecido pelo parâmetro *string2*.

Remove(string): Remove o elemento da lista de variáveis de sessão, cujo item possui nome de variável correspondente ao parâmetro string. Retorna verdadeiro se operação bem sucedida.

Remove(TPSVar): Remove o próximo elemento da lista de variáveis de sessão referenciado pelo parâmetro *TPSVar*. Retorna o item *TItemSVar* referente ao elemento removido.

Clear: Libera todos os ponteiros referentes à lista de variáveis de sessão. Reinicializa também a lista e atributos do objeto.

IsEmpty: Verifica se a lista de variáveis de sessão está vazia. Retorna verdadeiro caso esteja.

GetSessionVariables: Retorna os valores de todas as variáveis de sessão constantes na lista em uma única string de formato semelhante a "<nome_var1> = <valor1>; <nome_var2> = <valor2>".

Antes de executar qualquer método correspondente a uma requisição do cliente, ocorre um teste de expiração da sessão. Este teste é feito através do método *SessionExpired*.

A seguir, a definição da classe no código *Object Pascal*, junto com os seus tipos construídos de variáveis, estes últimos especificados na cor cinza:

```
 TSessionVariable = class (TObject)
   private
     First: TPSVar;
     Last: TPSVar;
     Sz: integer;
     Id: string;
     AppName: string;
     ExpirationTime: TDateTime;
     LastAccess: TDateTime;
     BDConnections: TConnectionVariable;
   protected
     function Remove(pAnt: TPSVar): TItemSVar; overload;
     function FindPos(Name:string; var pAnt:TPSVar): TPSVar;
     function GetSessionVariables: string;
   public
     constructor Create;
     procedure SetAppId(ApplicationName,SessionId: string);
     function SessionExpired: boolean;
     procedure AddTimeToExpire(TimeToExpire:TDateTime);
     procedure SetTimeToExpire(TimeToExpire:TDateTime);
     function Add(Item:TItemSVar): boolean;
     function Find(Name:string): TItemSVar;
     function Update(Name,NewValue:string): boolean;
     function Remove(Name:string): boolean; overload;
     procedure Clear;
     function IsEmpty: boolean;
     property Size: integer read Sz;
     property SessionId: string read Id;
     property ApplicationName: string read AppName;
     property Connections: TConnectionVariable read  BDConnections
write BDConnections;
     property Variables: string read GetSessionVariables;
   end;
```

DESENVOLVIMENTO E DISTRIBUIÇÃO (DEPLOYMENT)

O *deployPPW* é o programa distribuidor de aplicações PPW. É ele que tem a função de compilar, preparar e distribuir (*deployment*) a aplicação PPW, disponibilizando-a para execução. Ele converte as páginas PPW em executáveis e leva toda a estrutura de pastas da aplicação ao servidor PPW, em paralelo criando uma URL para a aplicação PPW.

As páginas PPW, que são as páginas HTML dinâmicas da tecnologia, possuem o formato texto e podem ser criadas e editadas em qualquer editor de textos ou de páginas HTML. Elas possuem basicamente a apresentação e formato como o exemplo mostrado na Figura 1.

```
<html>
<head>    <meta http-equiv="Content-Type" content="text/html">
<title>Teste Hello Word</title>
</head>
<body text="#000000" bgcolor="#FFFFFF" link="#0000FF" vlink="#000080" alink="#FF0000">
<font color="#000033" face="Arial, Helvetica, sans-serif">
<%var i: integer;%>
<%  for i:=1 to 10 do
    begin
%>
        HELLO WORLD!! <br>
<%
    end; %>
</font>
</body>
</html>
```

Figura 1. Exemplo básico de página PPW

Notem o uso de *scriptlets* delimitando os códigos em Pascal, tal qual acontece em tecnologias semelhantes como o ASP, PHP ou JSP.

Para cada aplicação PPW existe um único XML de configuração, chamado de *apw.xml*, que contém:

- Nome da aplicação;
- Os bancos de dados que podem ser conectados à aplicação, se necessário.

O *apw.xml* possui a seguinte estrutura básica, como mostra a Figura 2.

```
<appname>cadastro</appname>
<dbconn name="sgci">
    <dbdriver>MSSQL</dbdriver>
    <HostName>GER-WERTHER\WERTHER</HostName>
    <Database>SGCI</Database>
    <User_Name>sa</User_Name>
    <Password></Password>
</dbconn>
<dbconn name="sead">
    <dbdriver>MySQL</dbdriver>
    <HostName>localhost</HostName>
    <Database>sead_legislacao</Database>
    <User_Name>root</User_Name>
    <Password></Password>
</dbconn>
```

Figura 2. Exemplo do arquivo APW.XML

Resumidamente expondo, a tag *appname* fornece o nome da aplicação, que será também o seu alias no servidor HTTP. A tag *dbconn* visa identificar e especificar os bancos de dados que serão acessados pela aplicação, existindo uma para cada banco de dados que a aplicação se conecte.

O desenvolvimento de aplicações PPW é muito simples. Serão apresentadas a seguir algumas regras necessárias ao seu bom desenvolvimento.

Antes de tudo, deve-se criar uma pasta específica para um determinado projeto, colocando abaixo dele todas as sub-pastas que forem necessárias ao seu funcionamento, como se estivesse em um web site. A localização desta pasta principal fica ao critério do programador.

As páginas dinâmicas, ou seja, as páginas PPW, conforme visto antes, são páginas HTML intercalados com *scriptlets* contendo código *Object Pascal*. Todo comando, ou seqüências de comandos *Object Pascal*, deve estar sempre entre *scriptlets*. É imprescindível que cada *scriptlet* que abre – `<%` –corresponda a um que fecha – `%>`. Não são permitidos *scriptlets* intercalados.

Cada página deve conter no máximo, e quando houver necessidade, uma declaração *uses*, uma declaração *type*, uma declaração *const* e uma declaração *var*. Cada uma delas, se houver (não é obrigatório), tem que ser única, porém podem suportar múltiplas declarações. E para cada declaração *uses*, *type*, *const* ou *var*, é obrigatório o uso de um par exclusivo de *scriptlets* para cada. Ver exemplo de utilização de *scriptlets* em declarações na Figura 3. Os trechos (1), (2) e (3) mostram a utilização da declaração *uses*, *type* e *var* respectivamente, de acordo com a sintaxe do Pascal. Note que, apesar de serem contíguos, existe um par de *scriptlets* para cada.

Não é permitida a criação de *procedures* ou *functions* dentro das páginas. Caso seja necessário, deve-se criar uma *unit* pascal (.pas) e especificar a sua utilização na declaração *uses* da página PPW, como mostrado também na Figura 3.

```
...
<body text="#000000" bgcolor="#FFFFFF" link="#0000FF" vlink="#000080" alink="#FF0000" class="padrao">
<center>
  <p> <span class="padrao"><font size="+3">Resultado do Login</font> </span></p>
<% uses helptela in 'helptela.pas', teste in 'teste.pas'; %>
<%
   type
      TPCookie = ^TNodeCookie;
      TItemCookie = record
                     Name: string;
                     Value: string;
                   end;
%>
<% var i: integer; qry: TSQLQuery; %>
<%
    qry.Close;
    qry.SQLConnection := Session.getConnection('bdteste');
    if not Parameter.IsEmpty then
    begin
        if (Parameter.Value['nome']<>'') and (Parameter.Value['valor']<>'') then
        begin
            Session.Add(Parameter.Value['nome'],Parameter.Value['valor']);
%>
...
```

Figura 3. Exemplo de utilização de cláusulas em página PPW

Existem três objetos, fundamentais para a tecnologia, previamente instanciados e disponíveis para utilização do programador.

Tabela 2. Objetos PPW

Objeto	Classe	Descrição
Parameter	*TRequestParameter*	Armazena e disponibiliza todos os parâmetros recebidos pela página, via POST ou GET.
Session	*TClientSession*	Armazena e disponibiliza todas as informações da sessão e das variáveis da sessão, inclusive conexões a banco de dados, que são tratadas como variáveis de sessão.
Cookie	*TCookie*	Armazena e disponibiliza as informações sobre os *cookies*, permitindo manipular valores e datas de expiração.

As descrições das classes que pertencem estes objetos, juntamente com seus métodos, já foram abordadas em Componentes. Alguns exemplos de utilização destes objetos estão na Figura 3 e na Figura 4. Na primeira figura, os trechos marcados com (4) e (5) ilustram a utilização do objeto *Session*. No trecho (4) ele solicita uma conexão de banco com "`bdteste`" e no (5) adiciona uma variável de sessão. No trecho (5) também podem ser visto exemplos de utilização do objeto *Parameter*, onde os parâmetros de nome "`nome`" e "`valor`" são capturados.

```
<%
    if not Parameter.IsEmpty then
        Cookie.Add(Parameter.Value['nome'],Parameter.Value['valor'],Parameter.Value['exp']);
%>
<%
    Writeln('<B><BR><BR>COOKIES existentes: </B>');
    Writeln(GetEnvironmentVariable('HTTP_COOKIE'));
%>
...
  <p>Vari&aacute;vel <%=Parameter.Value['nome']%> possui o valor
  "<%=Session.GetValue(Parameter.Value['nome'])%>"</p>
<%
        end;
        sTempo_seg := Parameter.Value['tempos'];
        if sTempo_seg<>'' then
        begin
            nTempo_dias := StrToInt(sTempo_seg);
            Session.SetTimeToExpire(nTempo_dias);
%>
```

Figura 4. Exemplo de utilização dos objetos *Parameter*, *Cookie* e *Session*.

Existem mais classes e funções disponíveis que já vem nas bibliotecas específicas da tecnologia. Elas também foram abordadas em Componentes.

As páginas PPW, e qualquer arquivo do pertencente à aplicação, podem ficar em quaisquer pastas, desde que estejam abaixo ou no diretório principal da aplicação. Exceção se faz nas *units* do *Object Pascal*, que podem ficar em qualquer lugar, pois não vão para o servidor de aplicação. Elas só são utilizadas no processo de compilação das páginas.

É importante não esquecer do arquivo *apw.xml*, que tem que estar no diretório principal da aplicação. Nele estão especificados o nome da aplicação (que também serve como *alias* de acesso no servidor PPW) e os bancos a serem conectados. Como foi utilizado a solução *dbExpress* do *Delphi 7* para acesso a banco, a tecnologia PPW as mesmas capacidades e limitações nestas condições. É possível estabelecer conexões diretas com os bancos *Oracle, MS-SQL, MySQL, DB2, Interbase* e *Informix*.

Configuração do Arquivo APW.XML

O *apw.xml* é constituído de duas tags e suas subtags derivadas. A tag *appname* fornece o nome da aplicação, que será também o seu *alias* no servidor PPW. Não é permitida mais de uma aplicação com o mesmo nome. A sua definição é bem simples, pois ela não apresenta subtags. Sua estrutura é a seguinte:

```
<appname>nome_da_aplicação</appname>
```

Depois da aplicação distribuída e instalada no servidor PPW, para chamar a aplicação basta digitar e chamar no navegador *http://<url_do_servidor>/<nome_da_aplicação>*.

As tags *dbconn* de uma aplicação (pode ter mais de uma) fornece informações para que a aplicação estabeleça conexão com o(s) banco(s) desejado(s). Na definição da tag, o atributo *name* indica o nome interno da conexão que será usado em toda a aplicação PPW:

```
<dbconn name="nome_interno_conexao">
       . (espaço para subtags)
       . (espaço para subtags)
</dbconn>
```

A primeira subtag de *dbconn* informa o *driver* de conexão a ser utilizado, de acordo com o banco de dados:

```
<dbdriver>identificacao_do_banco</dbdriver>
```

Podem existir os seguintes valores para `identificacao_do_banco`:

Tabela 3. Valores da tag *dbdriver* para bancos

Valor	Banco de Dados Correspondente
MySQL	MySQL 3.23 e 4.0
MSSQL	MS-SQL 7.0 e 2000
DB2	IBM DB2 7.0
Oracle	Oracle 9i
Interbase	Interbase 6.5
Informix	Informix 9.2.1

Mais uma vez é conveniente lembrar que os *drivers* oferecidos e bancos correspondentes são provenientes do *dbExpress* do *Delphi 7*.

O restante das subtags obedece às seguintes regras:

Para configurar o acesso ao *DB2* e ao *Oracle*, basta definir as subtags:

```
<Database>nome_do_banco</Database>
<User_Name>nome_do_usuario</User_Name>
<Password>senha_correspondente_ao_usuario</Password>
```

Para configurar o acesso ao *MS-SQL*, *MySQL* e *Informix* é necessário acrescentar a subtag:

```
<HostName>nome_do_host</HostName>
```

Ao *Interbase*, basta acrescentar a seguinte subtag:

```
<RoleName>nome_do_perfil</RoleName>
```

Estas são as configurações mínimas e usuais para definição do arquivo *apw.xml*. Foi mostrado anteriormente na Figura 2 um exemplo de configuração do arquivo.

Para mais opções de acesso a banco, é só verificar as opções existentes no arquivo *dbxdrivers.xml*, que está no lado do servidor, situado na pasta *conf* do servidor PPW. A estrutura deste arquivo é exibida na Figura 5.

```xml
<dbxdrivers>
<dbdriver name="MySQL">
    <GetDriverFunc>getSQLDriverMYSQL</GetDriverFunc>
    <LibraryName>dbexpmys.dll</LibraryName>
    <VendorLib>libmysql.dll</VendorLib>
    <HostName></HostName>
    <Database></Database>
    <User_Name></User_Name>
    <Password></Password>
    <BlobSize>-1</BlobSize>
    <ErrorResourceFile></ErrorResourceFile>
    <LocaleCode>0000</LocaleCode>
</dbdriver>
<dbdriver name="MSSQL">
    <GetDriverFunc>getSQLDriverMSSQL</GetDriverFunc>
    <LibraryName>dbexpmss.dll</LibraryName>
    <VendorLib>oledb</VendorLib>
    <HostName></HostName>
    <Database></Database>
    <User_Name></User_Name>
    <Password></Password>
    <BlobSize>-1</BlobSize>
    <LocaleCode>0000</LocaleCode>
    <MSSQL-TransIsolation>ReadCommited</MSSQL-TransIsolation>
    <OS-Authentication>False</OS-Authentication>
</dbdriver>
<dbdriver name="DB2">
    <GetDriverFunc>getSQLDriverDB2</GetDriverFunc>
    <LibraryName>dbexpdb2.dll</LibraryName>
    <VendorLib>db2cli.dll</VendorLib>
    <Database></Database>
    <User_Name></User_Name>
    <Password></Password>
    <BlobSize>-1</BlobSize>
    <ErrorResourceFile></ErrorResourceFile>
    <LocaleCode>0000</LocaleCode>
    <DB2-TransIsolation>ReadCommited</DB2-TransIsolation>
</dbdriver>
<dbdriver name="Oracle">
    <GetDriverFunc>getSQLDriverORACLE</GetDriverFunc>
    <LibraryName>dbexpora.dll</LibraryName>
    <VendorLib>oci.dll</VendorLib>
    <Database></Database>
    <User_Name></User_Name>
    <Password></Password>
    <RowsetSize>20</RowsetSize>
    <BlobSize>-1</BlobSize>
    <ErrorResourceFile></ErrorResourceFile>
    <LocaleCode>0000</LocaleCode>
    <Oracle-TransIsolation>ReadCommited</Oracle-TransIsolation>
</dbdriver>
<dbdriver name="Interbase">
    <GetDriverFunc>getSQLDriverINTERBASE</GetDriverFunc>
    <LibraryName>dbexpint.dll</LibraryName>
    <VendorLib>gds32.dll</VendorLib>
    <Database></Database>
    <RoleName></RoleName>
    <User_Name></User_Name>
    <Password></Password>
    <ServerCharSet></ServerCharSet>
    <SQLDialect>1</SQLDialect>
    <BlobSize>-1</BlobSize>
    <CommitRetain>False</CommitRetain>
    <WaitOnLocks>True</WaitOnLocks>
    <ErrorResourceFile></ErrorResourceFile>
    <LocaleCode>0000</LocaleCode>
    <Interbase-TransIsolation>ReadCommited</Interbase-TransIsolation>
</dbdriver>
<dbdriver name="Informix">
    <GetDriverFunc>getSQLDriverINFORMIX</GetDriverFunc>
    <LibraryName>dbexpinf.dll</LibraryName>
    <VendorLib>isqlb09a.dll</VendorLib>
    <HostName></HostName>
    <Database></Database>
    <User_Name></User_Name>
    <Password></Password>
    <BlobSize>-1</BlobSize>
    <ErrorResourceFile></ErrorResourceFile>
    <LocaleCode>0000</LocaleCode>
    <Informix-TransIsolation>ReadCommited</Informix-TransIsolation>
</dbdriver>
</dbxdrivers>
```

Figura 5. O arquivo dbxdrivers.xml

Depois de construída as páginas e estabelecido corretamente as configurações do *apw.xml*, o próximo passo é executar o *deployPPW* para a aplicação ser preparada e distribuída em um servidor PPW.

O Processo de Distribuição PPW

Ao executar o *deployPPW*, o comando tem que ser executado da pasta do projeto da aplicação, ou seja, tem que ser executado no diretório principal da aplicação que está sendo desenvolvida. É necessária a utilização de uma variável de ambiente chamada de PPW_HOME que aponta para o diretório principal do servidor PPW. Outras variáveis de ambiente necessárias são PASCAL_HOME que aponta para o diretório principal do compilador *Object Pascal* e o PASCAL_COMPILER que contém o nome do programa-compilador. Apesar de hoje o PPW estar vinculado ao *Delphi 7*, tais variáveis de ambiente visam uma portabilidade futura para outros compiladores, além do *Delphi 7*.

Para executar o *deployPPW* basta digitar, no diretório principal da aplicação, *deployPPW* seguido do nome da aplicação. Ver exemplo da Figura 6. Neste exemplo, foi distribuída com sucesso a aplicação “login” no servidor PPW, utilizando como suporte o compilador do *Delphi 7*. É possível notar e distinguir as mensagens do compilador e as mensagens de controle da aplicação.

```
C:\MRedes\Tese\Workspace\App1_Login>deployppw login
Getting environment variables...
Converting PPW files...
Borland Delphi Version 15.0
Copyright (c) 1983,2002 Borland Software Corporation
index.ppw.pas(49)
50 lines, 0.29 seconds, 318248 bytes code, 11113 bytes data.
Borland Delphi Version 15.0
Copyright (c) 1983,2002 Borland Software Corporation
reslogin.ppw.pas(54)
55 lines, 0.30 seconds, 318524 bytes code, 11113 bytes data.
Creating PPW files from APW definition...
Creating application...
Cleaning environment...
Cleaning environment...
Application deployment successfully!
```

Figura 6. Execução típica do DeployPPW

O processo de distribuição, gerenciado pelo *deployPPW*, executa as seguintes etapas, conforme fluxograma simplificado ilustrado na Figura 7.

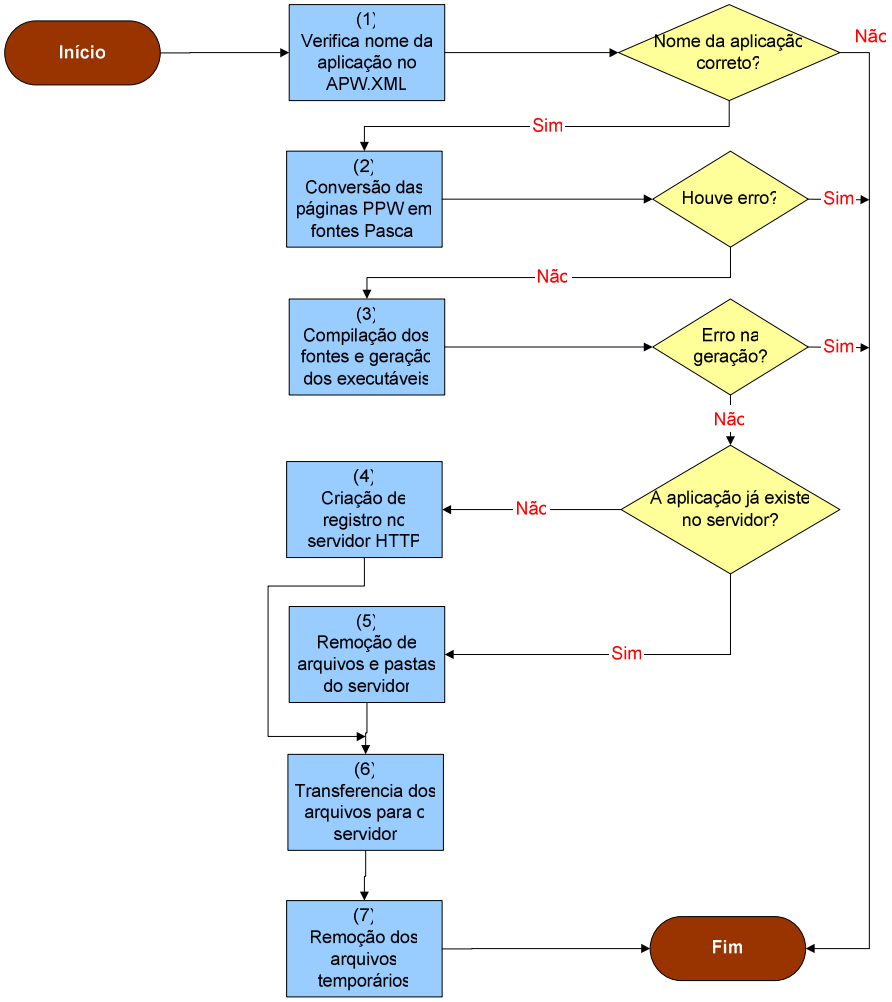

Figura 7. Fluxograma simplificado da execução do DeployPPW

(1) Verifica se o nome da aplicação está correto, se corresponde ao nome que está no arquivo *apw.xml*;

(2) Converte todas as páginas PPW (.ppw), inclusive as que estiverem em sub-pastas dentro da aplicação, em programas fontes Pascal (.pas);

   a. Na conversão de páginas PPW em fontes Pascal são adicionadas linhas de código essenciais ao uso da tecnologia e de seus objetos disponibilizados. Essas linhas servem apenas para auxiliar e completar as funcionalidades das páginas, nunca alterá-las na sua essência.

(3) Compila todos os programas fontes Pascal gerados, gerando executáveis, um para cada programa fonte;

   a. Em caso de erro, a mensagem é apresentada em tela e o programa fonte (.pas) que gerou o erro não é apagado; fica disponível para melhor inspeção e depuração posterior.

(4) Se tudo correu bem e a aplicação ainda não existir antes, é criado um registro no arquivo de configuração do servidor HTTP especificando o *alias* da aplicação e criado uma pasta específica para ela no servidor PPW;

(5) Se tudo correu bem, mas a aplicação já existia previamente, ou seja é apenas uma atualização da mesma, o diretório principal da aplicação e todo o seu conteúdo no servidor PPW é completamente removido e é criada novamente uma pasta específica para a aplicação no servidor PPW;

(6) Todos os arquivos do diretório principal de desenvolvimento e todas as suas sub-pastas são trazidos para o servidor e colocados na pasta específica da aplicação;

(7) No final, são removidos automaticamente todos os arquivos temporários do ambiente de desenvolvimento (.exe) e do servidor (.pas, .dcu, .obj) gerados no processo;

Todo o processo é controlado, etapa por etapa. Caso ocorra algum problema, é sempre apresentada uma mensagem de erro clara, personalizada e vinculada à etapa que ocorreu o problema.

Caso não apresente erro no processo, após o seu final a aplicação está distribuída e disponível no servidor PPW em *http://<url_do_servidor>/<nome_da_aplicação>*.

SERVIDOR PPW

É chamado de servidor PPW a parte *server-side* que gerencia o processamento da aplicação PPW requisitada. Uma representação adequada do seu funcionamento é mostrada na Figura 8.

O *PPWController* trabalha do lado de servidor (*server-side*), sendo portanto componente fundamental da tecnologia PPW. É responsável pelo gerenciamento de sessões, variáveis de sessão e conexões a bancos de dados. Está sempre ativo e residente na memória ouvindo a porta 81 do servidor. Ele fica respondendo, em modo *multi-thread*, à execução dos programas PPW (executáveis CGI) quando são chamados pela aplicação. Ele é o responsável pelo controle de sessões, podendo incluir, remover ou alterar informações e variáveis de sessão. É também o responsável pelo estabelecimento e manutenção de conexões aos bancos de dados e a disponibilização e distribuição delas para a aplicação. Funciona como uma espécie de repositório de sessões e controlador da aplicação PPW, no coração da arquitetura *server-side*. A Figura 8 ilustra o funcionamento do *PPWController* junto a arquitetura do servidor PPW, visando um melhor entendimento.

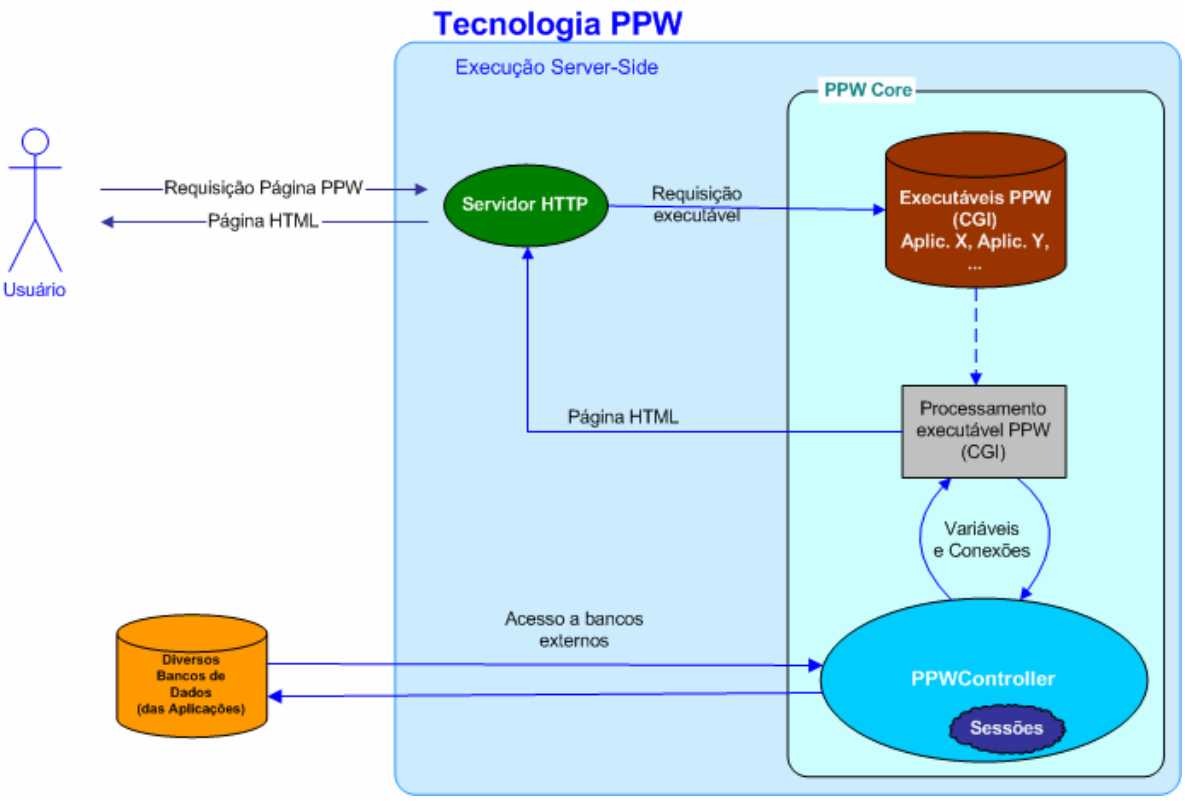

Figura 8. Arquitetura do servidor PPW

O servidor PPW possui uma distribuição de pastas, como mostrada na Figura 9. É necessária a configuração da variável de ambiente PPW_HOME, apontando para o diretório principal do servidor PPW.

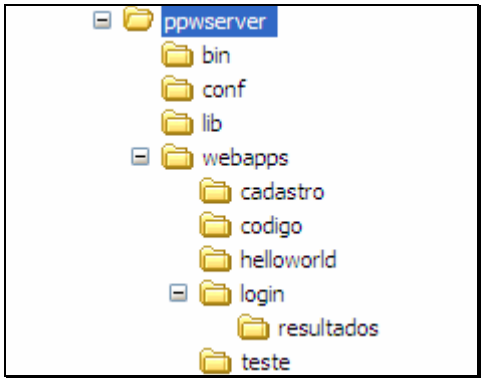

Figura 9. Estrutura do servidor PPW

Abaixo da pasta *webapps* estarão as pastas para cada aplicação. Cada aplicação possui uma pasta principal e pode conter zero ou mais sub-pastas.

O servidor PPW trabalha em uma estrutura de diretórios independentes ao do servidor HTTP utilizado, no caso, o Apache. A idéia é, em futuro próximo, torná-lo independente de servidor HTTP e facilitar a sua flexibilidade em usar outros servidores HTTP, pois bastaria substituir as rotinas de configuração do servidor Apache, muitas feitas diretamente pelo programa de distribuição, o *deployPPW*, pelas do servidor HTTP desejado. Mas atualmente ele trabalha junto ao servidor HTTP Apache, conforme mencionado antes.

Na pasta *bin*, existem os arquivos de execução necessários ao servidor PPW, o deployPPW.exe e o PPWController.exe conforme mostrados na Figura 10.

```
C:\ppwserver\bin>dir
O volume na unidade C não tem nome.
O número de série do volume é 486B-629E

Pasta de C:\ppwserver\bin

23/05/2005  04:31    <DIR>          .
23/05/2005  04:31    <DIR>          ..
13/01/2005  14:14           194.048 DeployPPW.exe
13/01/2005  14:14           562.176 PPWController.exe
2 arquivo(s)        756.224 bytes
2 pasta(s)  5.456.748.544 bytes disponíveis
```

Figura 10. Conteúdo de PPW/Bin

Na pasta *conf* existe apenas o arquivo *dbxdriver.xml*, responsável pelo padrão de configuração de acesso para cada banco, cuja estrutura já foi mostrada anteriormente na Figura 5.

Na pasta *lib* existem bibliotecas comuns à tecnologia e que também disponibilizam suas funções e classes, apresentadas em

Componentes. Podem ser adicionadas também outras bibliotecas, se assim for conveniente ao programador.

Iniciando Servidor PPW

Para o funcionamento do servidor PPW, o programa *PPWController* tem que estar em operação, junto, é claro, com o servidor HTTP. O *PPWController* é o responsável pelo controle de sessões e conexões, como já explicado antes. Funciona como um repositório central de informações para a aplicação/cliente. Caso o *PPWController* não esteja no ar, após uma tentativa de requisição é exibida uma mensagem de erro semelhante à Figura 11.

Quando o servidor PPW está no ar, ele trabalha com um objeto *Sessions*, que pertence à classe *TList*. A classe *TList*, já proveniente do *Delphi 7*, representa uma lista (ou coleção) on-line de objetos. No caso do objeto *Sessions*, ele guarda uma lista de todas as sessões web ativas no servidor. Cada objeto da lista é da classe *TSessionVariable*, cuja estrutura e métodos foi visto antes em TSessionVariable. Cada objeto corresponde a uma sessão web de uma determinada aplicação PPW, guardando todas as informações referentes à sessão, inclusive conexões e variáveis. Ele é identificado pelo número de identificação da sessão e a aplicação PPW correspondente.

Para uma aplicação PPW iniciar é necessário a chamada a uma página PPW. Quando isto acontece, o servidor HTTP chama o aplicativo CGI correspondente à página PPW, que possui o mesmo nome da página, apenas acrescentado da extensão *.exe*. O aplicativo chamado é executado e, se necessário for para sua execução, captura as informações de parâmetros passadas através do GET e POST utilizando o objeto *Parameter*, conforme explicado anteriormente. O aplicativo CGI em execução, doravante chamado de aplicativo PPW, pode também interagir com o *PPWController* através de três tipos de requisições: sessão, variável de sessão e variável de conexão a banco de dados. Uma ilustração deste funcionamento é observada na Figura 8.

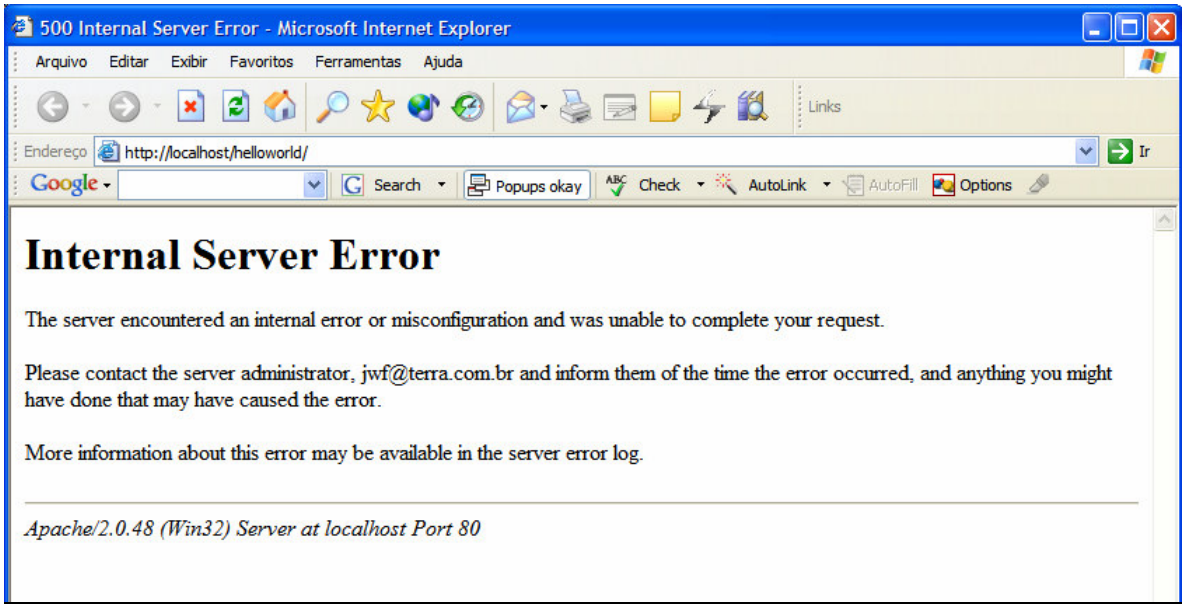

Figura 11. Problema de acesso ao servidor PPW

Requisições ao Servidor PPW

A requisição do **tipo sessão** possui três subtipos: nova sessão, alteração tempo de expiração da sessão e de validação da sessão. As requisições e os respectivos retornos são feitos e tratados pelo objeto *Session*.

A requisição de nova sessão fornece um novo número de sessão ao cliente. Utilizado automaticamente quando o cliente faz a sua primeira requisição ao servidor. Existe uma rotina para geração de um novo número de sessão no *PPWController*. A identificação da sessão é armazenada por um *cookie* de sessão na máquina cliente.

A alteração do tempo de expiração refere-se à alteração do tempo limite de não requisição ao servidor para que a sessão continue válida. O tempo padrão de expiração de sessão é de 30 (trinta) minutos quando a sessão é criada.

A requisição de validação de sessão testa se a sessão ainda é válida, analisando especificamente o tempo de expiração. Se o tempo entre requisições é maior do que o tempo marcado para expiração, a sessão é invalidada. Possui também a requisição que faz com que a sessão seja invalidada automaticamente, alterando para isso o tempo de expiração para -1.

A requisição do **tipo variável de sessão** possui quatro subtipos: criação de variáveis, alteração de valores de variáveis, remoção da variável e consulta ao valor da variável. Esses valores são armazenados em um objeto *TSessionVariable* que pertence ao objeto *Sessions* do *PPWController* de acordo com o número de identificação da sessão e respectiva aplicação. Como mencionado anteriormente, o objeto *Sessions* representa uma coleção de objetos *TSessionVariable*.

A requisição do **tipo variável de conexão a banco de dados** funciona da seguinte maneira: quando a requisição é feita, o *PPWController* tenta estabelecer uma conexão com o banco de dados, conforme configuração informada no *apw.xml* da aplicação. Caso consiga, a variável de conexão obtida é armazenada no *PPWController*., no objeto *TSessionVariable* da coleção *Sessions*, referente a sessão que o requisitou. Depois é convertida em string e enviada ao aplicativo PPW requisitante, que a reverte em um objeto de conexão através do objeto *Session*. Em toda requisição de uma mesma conexão por uma mesma sessão (aplicação/cliente) ao servidor, a conexão não é refeita, pois a variável de conexão já existe armazenada no *PPWController*. Ela é apenas enviada para o requisitante. A variável de conexão só pode ser criada quando ainda não existe para uma determinada sessão.

Caso a conexão não consiga ser estabelecida por qualquer motivo, uma exceção é levantada pelo *PPWController* e passada ao aplicativo PPW que informa o problema na tela do navegador HTTP, como mostrado na Figura 12.

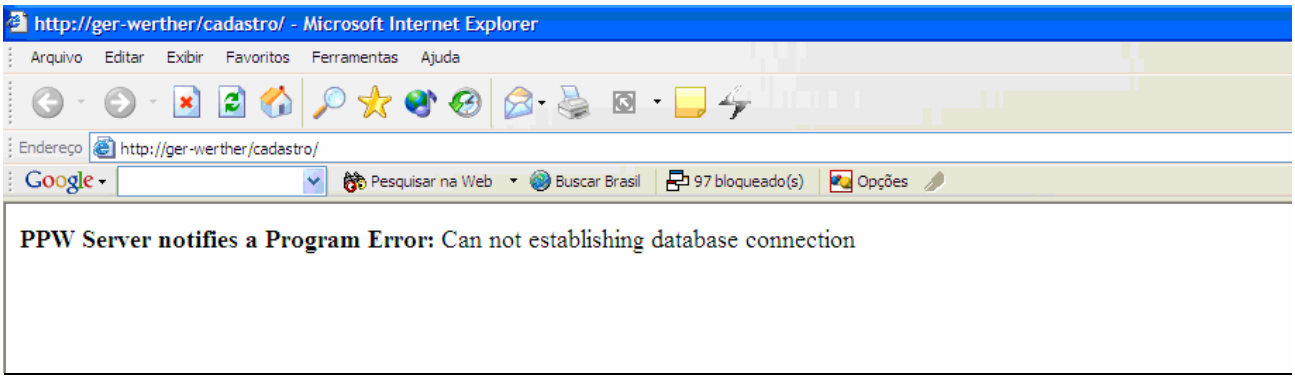

Figura 12. Problemas na conexão do servidor PPW ao banco de dados

Aliás, qualquer erro do aplicativo PPW resulta em uma mensagem na tela do navegador que o requisitou. Existe uma grande variedade de mensagens de erro, a maioria gerada pelo compilador do *Delphi 7*. As mensagens geradas pela tecnologia PPW tem origem no *PPWController* e são as seguintes:

Session was expired: quando o tempo da sessão expirou;

Session variable already exists: quando deseja criar uma variável já existente na sessão;

Session variable does not exists: quando se deseja obter o valor de uma variável de sessão inexistente;

Can not remove session variable: quando tenta remover uma variável de sessão inexistente;

Can not update session variable: quando tenta atualizar o valor de uma variável de sessão inexistente;

Can not establishing database connection: quando não se consegue estabelecer a conexão com o banco de dados, conforme explicado anteriormente;

Internal Error: Invalid session-id or application name: quando o identificador da sessão é nulo ou não está associado a nenhuma aplicação – é apenas preventivo.

Tratamento de Cookies

Todos os *cookies* da aplicação existentes na máquina cliente são capturados pelo servidor PPW, deixando todos disponíveis nas páginas PPW através do objeto *Cookie*. É possível a qualquer momento criar um novo *cookie* de sessão ou com uma determinada data de expiração. Mais detalhes de implementação das funcionalidades do objeto *Cookie*, em TCookie. Lembrando apenas que o *cookie* só é criado de fato na máquina cliente depois da submissão da página PPW que requisitou sua criação. É bom ressaltar que a geração efetiva de *cookies* está condicionada às definições de privacidade do navegador HTTP do cliente.