Client/Server Caching Using File Streams

by Neil McClements

Despite the current vogue for thin client applications, there's still life left in the thick client yet. Plenty of applications have to contend with limited network bandwidth, an over-stretched data server and conflicting processing requirements. This article explores ways of easing the strain. We'll see how components can be developed to cache both themselves and their data locally. With client caching in place, the data server can spend less time servicing routine client requests. Network bandwidth can be clawed back for essential traffic, too. And users should find their applications more responsive and configurable.

Data Caching

Two-tier client server systems typically consist of a single database which services multiple clients. Though the server is optimised for data manipulation, a collection of PCs will often represent considerably more processing clout. The difficulty lies in co-ordinating this power. Ideally, we should be able to reduce the number of times each application hits the database. This will cut down network traffic across a WAN/LAN and, in theory, should improve the performance of the applications themselves.

Data caching allows regularly retrieved data to be held locally on the client rather than on the database server. Perhaps the easiest and most obvious way to implement this approach is to hard-code common data into an application or its components. For instance, for a combo box which contains a set of currency codes the developer might write a fragment of code in a form create procedure like Listing 1.

This works just fine. However, there is always the possibility that your client will start trading with a Dutch partner. Adding Dutch Guilders to the list of currencies is simple enough in itself. However, this simple change requires the application to be rebuilt and redistributed. Again, this can be easy enough for a handful of client PCs. But how do we manage synchronised rapid updates to dozens or even hundreds of client PCs?

Much better, then, to hold list items like currency codes on a database. The list box would populate itself using a piece of code like Listing 2. Adding a currency to the list would entail inserting a new record on the currency table. Similarly, deleting or changing currency details would be accomplished by altering the records on the currency table.

Though storing items on the database is a step forward, the cost of such flexibility is performance. The data server is hit each time a currency combo box is populated. The list is refreshed from scratch each time, even if the currency details

Listing 1

```
MyCombo.items.add('GBP - Pounds Sterling');
MyCombo.items.add('FFr - French Franc');
MyCombo.items.add('USD - US Dollar');
```

Listing 2

```
procedure Populate;
var
  qry:TQuery;
  code, name:string;
begin
  qry:=TQuery.create;
  qry.sql.add('select currency_code, currency_name from currency');
  qry.open;
  qry.first;
  while not qry.eof do begin
    code := qry.fieldbyname('currency_code').asString;
    name := qry.fieldbyname('currency_name').asString
    MyCombo.items.add(code + ' - ' + name);
    qry.next;
  end;
  qry.free;
end;
```

Listing 3

```
procedure TCacheForm.btnSaveToStreamClick(Sender: TObject);
var
  stream:tfilestream;
begin
  stream := TFileStream.create('c:\test.cfg', fmCreate or fmOpenWrite);
  ComboBox1.items.savetostream(stream);
  stream.free;
end;
```

Listing 4

```
procedure TCacheForm.btnLoadFromStreamClick(Sender: TObject);
var
  stream:tfilestream;
begin
  stream := TFileStream.create('c:\test.cfg', fmOpenRead);
  combobox1.items.loadfromstream(stream);
  stream.free;
end;
```

haven't changed. Many applications can tolerate dozens of clients periodically refreshing their controls from the same server. However, there can be circumstances where it is preferable to minimise the volume of repetitive database requests and network traffic.

Data Shelf Life

In any single application there can be data of varying ages. Some data changes regularly throughout the day, for example account balances. Other data, like currency codes, has a very long shelf life. Caching can be used to good effect for data that tends to change rarely. Since there is a relatively consistent set of world currencies, it might make sense to store all the possibilities on the database. A currency list component can be refreshed once from the database. Thereafter, the component caches the details until they change and a full refresh is needed again.

Caching Using File Streams

File streams are just sequences of binary data held in a file. Objects or components can be dropped into a stream as a series of bytes. When the objects are retrieved, Delphi reconstitutes the byte stream in the correct object form, properties and all. Streams derive much of their power and flexibility from this object oriented approach. A handful of VCL objects have their own `SaveToStream` and `LoadFromStream` methods. For example, the contents of a combo box could be cached using code like Listing 3.

The stream is created with the target filename and file mode as parameters. The combo box items method `SaveToStream` writes each item to the file in turn. Reading the values back is done in a similar manner. This time, we need to change the file mode to `fmOpenread` and call the `TStrings` method `LoadFromStream`. See Listing 4. Choose your file mode carefully. Forgetting to change the mode from `fmOpenWrite` causes an empty stream to be recreated over the top of the existing cache file.

Caching can be taken further. In addition to recording the combo box items, we could record the selected combo item between application sessions. When the user started the application they would see the selection from last time. This provides a convenient means of storing a user's application preferences. For instance, a cached combo box could hold a user's preferred display font for other components in the same application.

Cached Combo Box

Caching behaviour can be added to a combo box by deriving a new `TCachedCombo` from `TComboBox`. The new component will provide a couple of extra methods: `ReadListFromStream` and `WriteListToStream`. The stream file will be identified using a `ConfigFile` property. For convenience, the new component is data aware. Items can be loaded from a `DataSource` identified by the `ItemsDataSource` property. The accompanying `ItemsDataField` property is used to pick the correct field for display.

The combo is filled initially by calling `Populate`. Thereafter, the component can be refreshed either from its cache or by reading data from the `DataSource` again. The `UseCache` property allows the developer to switch the component's mode from `DataSource` driven to cached.

The developer can enter the combo items manually at design time or, preferably, once from the `DataSource` to provide items to cache. The component code is shown in Listing 5.

The first step requires the definition of a `TStringList` wrapper to hold both the combo item list and the current selection. The `TComboList` exists to enable its published properties to be streamed to a file. By deriving the data structure from a `TComponent`, both published and public properties can be recorded in a stream.

The `TCachedCombo` registers the new object using `RegisterClass(TComboList)`. This ensures that when the time comes to retrieve the combo details, Delphi recognises the format of the objects in the file stream and reconstitutes them correctly.

Combo box details are cached using the `WriteListToStream` method. A temporary `TComboList` is created and the combo items are assigned along with the current itemindex. The stream is created as before, only this time the `TComboList` is written to the stream rather than just the combo box items.

Retrieving the cached data is achieved using `ReadListFromStream`. The stream file is opened and `Stream.ReadComponent(nil)` returns the component. At this point, the component could be anything: list box, string list or `TComboList`. Since the `TComboList` class has been registered in the cached combo's constructor the "mystery" component can be cast to a `TComboList`. It is then a simple matter to set the `TCachedCombo` items and `ItemIndex` to the retrieved `ItemValues` and `SelectedItem` properties.

Cached Calendar Component

This approach to data caching can be extended for more complex caches. Any cache can be implemented as a registered `TComponent` descendent. These objects can then be written to or read from streams. A `TCalendar` could be adapted to record significant dates like public holidays in a file stream. As the calendar was displayed, the recorded dates would be read from the stream. Notable dates could be shown in red and a user would be able to click on the date for a popup summary of the occasion.

Listing 6 shows the code for a cached, data aware calendar component. This component illustrates the significance of data shelf life in cache design. Typically, a calendar showing public holidays would only need to be refreshed from a database once per year.

The holidays are unlikely to change *within any one year*, so by caching dates, the component spares the host database from unnecessary refreshes each time the parent application is started. Of course, a more dynamic business calendar could be refreshed weekly or daily, perhaps showing delivery dates, invoice dates and so on. The component leaves the

```
unit CachedCombo;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, DBCtrls, DBTables, DB;
type
  { ComboList - holds the items and itemindex briefly whilst
    written to/retrieved from a stream }
  TComboList = class(TComponent)
  private
    FValues:TStringList;
    FItemIndex:longint;
  public
    constructor Create(AOwner:TComponent);override;
    destructor  Destroy; override;
  published
    property ItemValues:TStringList
      read Fvalues write Fvalues;
    property SelectedItem:longint
      read FItemIndex write FItemIndex;
  end;
  { TCachedCombo - allows the items to be filled from a
    datasource and cached }
  TCachedCombo = class(TComboBox)
  private
    FConfigFile:string;
    FItemsFieldDataLink:TFieldDataLink;
    FUseCache:boolean;
    function  GetItemsDataField:string;
    function  GetItemsDataSource:TDataSource;
    procedure ItemsDataChange(Sender:TObject);
    procedure SetItemsDataField(const theFieldName:string);
    procedure SetItemsDataSource(theSource:TDataSource);
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation);
    procedure SetUseCache(CacheOnOff:boolean);
  public
    constructor Create(Owner:TComponent);override;
    destructor  Destroy; override;
    function    Populate:boolean;
    function    ReadListFromStream:boolean;
    function    WriteListToStream:boolean;
  published
    property ConfigFile:string
      read FConfigFile write FConfigFile;
    property ItemsDataField:string
      read GetItemsDataField write SetItemsDataField;
    property ItemsDataSource:TDataSource
      read GetItemsDataSource write SetItemsDataSource;
    property UseCache:boolean
      read FUseCache write SetUseCache default false;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('more...', [TCachedCombo]);
end;
constructor TComboList.Create(AOwner:TComponent);
begin
  inherited Create(AOwner);
  Fvalues := TStringList.create;
end;
destructor TComboList.Destroy;
begin
  { Tidy up after the component before calling its
    ancestor's destructor }
  Fvalues.free;
  inherited destroy;
end;
constructor TCachedCombo.Create(Owner : TComponent);
var exePath : string;
begin
  inherited Create(Owner);
  { Prepare the data link }
  FItemsFieldDataLink:=TFieldDataLink.Create;
  FItemsFieldDataLink.OnDataChange := ItemsDataChange;
  { Default config file used to cache dates between
    application sessions }
  if FConfigFile = '' then begin
    exePath := ExtractFilePath(application.exename);
    FConfigFile := exePath+'Combo.cfg';
  end;
  { Register the date list component so Delphi knows how to
    handle it in file streams }
  RegisterClass(TComboList);
end;
destructor TCachedCombo.Destroy;
begin
  FItemsFieldDataLink.free;
  inherited Destroy;
end;
function TCachedCombo.WriteListToStream : boolean;
var stream : TfileStream;
    ComboList : TComboList;
begin
  { Write the combo items to the cache via a file stream }
  try
    ComboList := TComboList.create(Owner);
    ComboList.ItemValues.assign(self.items);
    ComboList.SelectedItem:=ItemIndex;
    stream := TFileStream.create(FConfigFile,
      fmCreate or fmOpenWrite);
    stream.WriteComponent(ComboList);
    stream.free;
    ComboList.free;
    Result := true;
  except
    on E : exception do
      Result := false;
  end; { except }
end; { function }
function TCachedCombo.ReadListFromStream:boolean;
var stream : TfileStream;
    ListComponent : TComponent;
begin
  try
    stream := TfileStream.create(FConfigFile, fmopenread);
    while not (stream.position = stream.size) do begin
      { Read the next component in the stream and try and
        determine what it is }
      ListComponent := stream.ReadComponent(nil);
      if (ListComponent is TComboList) then begin
        items.assign(
          (ListComponent as TComboList).ItemValues);
        self.ItemIndex :=
          (ListComponent as TComboList).SelectedItem;
      end;
    end;
    stream.free;
    Result := true;
  except
    on E : EFOpenError do Result:=false;
  end; { except }
end;
procedure TCachedCombo.SetUseCache(CacheOnOff : boolean);
begin
  { Refresh the combo box whenever the developer switches
    the component mode from cached to not-cached }
  if (CacheOnOff = True) then ReadListFromStream
  else Populate;
end;
function TCachedCombo.Populate : boolean;
var dSet : TDataset;
begin
  { Fill the combo box items string list from the datasource
    if the data link is still valid }
  if (assigned(FItemsFieldDataLink) and
    (ItemsDataSource<>nil)) then begin
    self.clear;
    dSet := FItemsFieldDataLink.DataSource.Dataset;
    dSet.Active := true;
    dSet.first;
    while not dSet.eof do begin
      self.items.add(dSet.FieldByName(
        FItemsFieldDataLink.FieldName).AsString);
      dSet.next;
    end;
    { Record updated details in the cache for next time... }
    Result := WriteListToStream;
  end else
    Result := false;
end;
{ following functions maintain datasource and data links references }
function TCachedCombo.GetItemsDataField : string;
begin
  GetItemsDataField := FItemsFieldDataLink.FieldName;
end;
function TCachedCombo.GetItemsDataSource : TDataSource;
begin
  GetItemsDataSource := FItemsFieldDataLink.DataSource;
end;
procedure TCachedCombo.SetItemsDataField(
  const theFieldName : string);
begin
  FItemsFieldDataLink.FieldName := theFieldName;
end;
procedure TCachedCombo.SetItemsDataSource(
  theSource : TDataSource);
begin
  FItemsFieldDataLink.DataSource := theSource;
end;
procedure TCachedCombo.ItemsDataChange(Sender : TObject);
begin
  if FItemsFieldDataLink.Field = nil then
    FUseCache := ReadListFromStream;
end;
procedure TCachedCombo.Notification(
  AComponent : TComponent; Operation : TOperation);
begin
  { If the datasource is removed from the application,
    reset the data source reference }
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (FItemsFieldDataLink <> nil)
    and (AComponent = ItemsDataSource) then
    ItemsDataSource := nil;
end;
end.
```

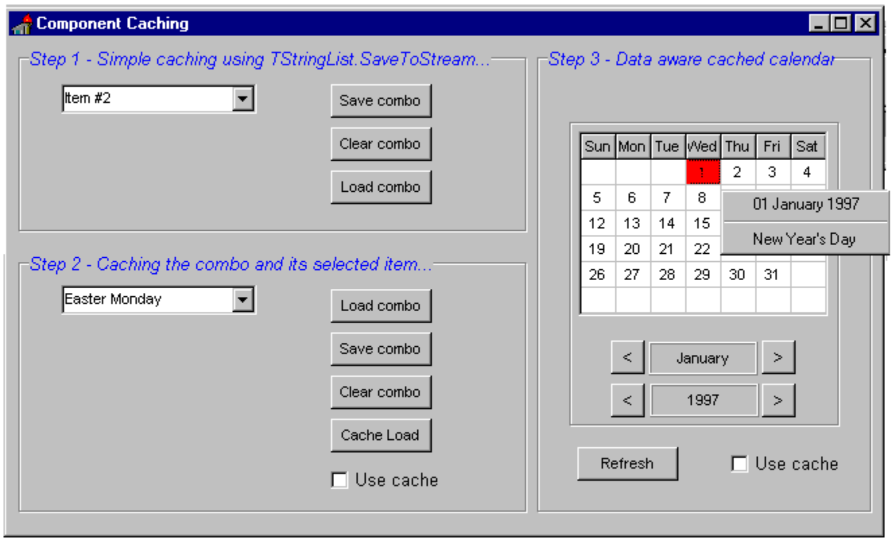
➤ *Facing page: Listing 5*

cache refresh period to the developer's discretion.

Once again, a wrapper component is created to store the significant date and a piece of text to act as a description of the occasion.

The significant dates are retrieved in the first instance from a query or table dataset linked to the `TDataSource` identified by the `DataSource` property. Two `TFieldDataLinks` are set up to take the date and occasion text from the `DataSource`. They are used to update the calendar's list of significant dates in the `Refresh` method. Once the dates have been retrieved from the `DataSource` they are immediately cached for later use.

The calendar itself is a descendent of `TCustomGrid`. In order to display the "red letter days" correctly, the `DrawCell` method has been overridden. Whenever a significant date is encountered, its cell is painted red by resetting the canvas brush and rewriting the day in the cell using `TextRect`. The component is forced to repaint itself after each `Refresh` or `ReadDatesFromStream` by calling `Invalidate`. This ensures that the modified `DrawCell` method fires and the calendar highlights the dates, even if the calendar hasn't already been redrawn.



Finally, the `Click` method has been modified to display a popup menu whenever a user selects a significant date. And that's it.

Bringing It All Together

The project TestCalendar on the disk shows both the cached combo and calendar in action. Typically, calendar dates would be held on a server. However, for the demo, a basic set of significant dates for the UK is included in the Paradox table CALENDAR.DB. At present, the calendar component assumes that each date represents only one occasion, but it could easily be extended to show multiple occasions against each date.

Caching can be a useful tool in the client/sever developer's armoury. As we've seen, data caching can enable database load to be partially transferred to the client making use of abundant PC resources. There is also potential for caches to maintain user preferences between application sessions.

Neil McClements is a consultant developer specialising in client/-server banking systems. Email Neil at nmcclements@pemail.net
© 1997 Neil McClements

➤ *Listing 6*

```
unit CachedCalendar;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar,DBCtrls, DBTables,DB,Menus;
type
  TSpecialDateList = Class(TComponent)
  private
    FDate : TDatetime;
    FOccasion : string;
  published
    property Date: TDatetime read FDate write FDate;
    property Occasion: string read FOccasion write FOccasion;
  end;
  TCachedCalendar = class(TCalendar)
  private
    FConfigFile : string;
    FUseCache : boolean;
    FDateFieldDataLink : TFieldDataLink;
    FTextFieldDataLink : TFieldDataLink;
    FDateList : TList;
    FDatePopupMenu : TPopupMenu;
    procedure DataChange(Sender : TObject);
    function  GetDataSource : TDataSource;
    function  GetDateField : string;
    function  GetTextField : string;
    procedure SetDataSource(theSource : TDataSource);
    procedure SetDateField(const theFieldName : string);
    procedure SetTextField(const theFieldName : string);
  protected
    procedure Click; override;
    procedure DrawCell(ACol, ARow :  Longint; ARect : TRect;
      AState : TGridDrawState); override;
    procedure Notification(AComponent : TComponent;
      Operation : TOperation);
    function  ReadDatesFromStream : boolean;
    procedure SetUseCache(CacheOnOff : boolean);
    function  WriteDatesToStream : boolean;
  public
    constructor Create(Owner : TComponent);override;
    destructor  Destroy; override;
    function    Refresh : boolean;
  published
    property ConfigFile : string
      read FConfigFile write FConfigFile;
    property DataSource : TDataSource
      read GetDataSource write SetDataSource;
    property DateField : string
      read GetDateField write SetDateField;
    property TextField : string
      read GetTextField write SetTextField;
    property UseCache : boolean
      read FUseCache write SetUseCache default false;
  end;
procedure Register;
implementation

procedure Register;
begin
  RegisterComponents('more...', [TCachedCalendar]);
end;

constructor TCachedCalendar.Create(Owner : TComponent);
var exePath : string;
begin
  inherited Create(Owner);
  { Configure two data links, one for the special date field
    and the other for the occasion text (eg "New Year's Day")
  FDateFieldDataLink := TFieldDataLink.Create;
  FTextFieldDataLink := TFieldDataLink.Create;
  FDateFieldDataLink.OnDataChange := DataChange;
  FTextFieldDataLink.OnDataChange := DataChange;
  { Default config file used to cache dates between sessions}
  if FConfigFile='' then begin
    exePath := ExtractFilePath(application.exename);
                                 { *** CONTINUED ON NEXT PAGE --> }
{ ** LISTING 6 CONTINUED FROM PREVIOUS PAGE }
   FConfigFile := exePath+'Calendar.cfg';
  end;
  RegisterClass(TSpecialDateList);
  FDateList := TList.Create;
  FUseCache := false;
end;
destructor TCachedCalendar.Destroy;
begin
  if assigned(FDateFieldDataLink) then
    FDateFieldDataLink.free;
  if assigned(FTextFieldDataLink) then
    FTextFieldDataLink.free;
  FDateList.free;
  inherited Destroy;
end;
procedure TCachedCalendar.SetUseCache(CacheOnOff : boolean);
begin
  if (CacheOnOff=true) then ReadDatesFromStream
  else Refresh;
end;
function TCachedCalendar.Refresh : boolean;
var
  dSet : TDataset;
  theDate : TDateTime;
  theOccasion : string;
  DateInfo : TSpecialDateList;
begin
  if (assigned(FDateFieldDataLink) and
    (DataSource<>nil)) then begin
    FDateList.clear;
    dSet := DataSource.Dataset;
    dSet.Active := true;
    dSet.first;
    while not dSet.eof do begin
      DateInfo := TSpecialDateList.create(Owner);
      DateInfo.Date := dSet.FieldByName(
        FDateFieldDataLink.FieldName).AsDatetime;
      DateInfo.Occasion := dSet.FieldByName(
        FTextFieldDataLink.FieldName).AsString;
      FDateList.add(DateInfo);
      dSet.next;
    end;
    Result := WriteDatesToStream;
  end else Result := false;
  Invalidate;
end;
function TCachedCalendar.WriteDatesToStream : boolean;
var
  stream : TfileStream;
  DateInfo : TSpecialDateList;
  c : longint;
begin
  try
    stream := TFileStream.create(
      FConfigFile, fmCreate or fmOpenWrite);
    for c := 0 to (FDateList.count-1) do begin
      DateInfo := FDateList.items[c];
      stream.WriteComponent(DateInfo);
    end;
    stream.free;
    Result := true;
  except
    on E : exception do Result := false;
  end; {except }
end; { function }
function TCachedCalendar.ReadDatesFromStream : boolean;
var stream : TfileStream;
    ListComponent : TComponent;
begin
  try
    FDateList.clear;
    stream := TfileStream.create(FConfigFile, fmopenread);
    while not (stream.position = stream.size) do begin
      ListComponent := stream.ReadComponent(nil);
      if (ListComponent is TSpecialDateList) then begin
        FDateList.add((ListComponent as TSpecialDateList));
      end;
    end;
    stream.free;
    Result := true;
  except
    on E : EFOpenError do Result := false;
  end; { except }
  Invalidate;
end;
function TCachedCalendar.GetDateField : string;
begin
  GetDateField := FDateFieldDataLink.FieldName;
end;
function TCachedCalendar.GetDataSource : TDataSource;
begin
  GetDataSource := FDateFieldDataLink.DataSource;
end;
procedure TCachedCalendar.SetDateField(
  const theFieldName : string);
begin
  FDateFieldDataLink.FieldName := theFieldName;
end;
procedure TCachedCalendar.SetDataSource(
  theSource :  TDataSource);
begin
  FDateFieldDataLink.DataSource := theSource;
end;
function TCachedCalendar.GetTextField : string;
begin
  GetTextField := FTextFieldDataLink.FieldName;
end;
procedure TCachedCalendar.SetTextField(
  const theFieldName : string);
begin
  FTextFieldDataLink.FieldName := theFieldName;
end;
procedure TCachedCalendar.DataChange(Sender : TObject);
begin
  if FDateFieldDataLink.Field = nil then
    FUseCache := ReadDatesFromStream;
end;
procedure TCachedCalendar.Notification(
  AComponent :  TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (FDateFieldDataLink <> nil)
    and (AComponent = DataSource) then DataSource :=  nil;
end;
procedure TCachedCalendar.DrawCell(ACol, ARow : Longint;
  ARect :  TRect; AState : TGridDrawState);
var
  CellValue :  string;
  CellDate : TDatetime;
  SearchDate : TSpecialDateList;
  d : longint;
begin
  inherited;
  CellValue := CellText[Acol, ARow];
  if ((CellValue<>'') and (ARow<>0)) then begin
    try
      CellDate :=
        encodedate(Year, Month, StrToInt(CellValue));
      for d := 0 to (FDateList.count-1) do begin
        SearchDate := FDateList[d];
        if (SearchDate.Date=CellDate) then begin
          { When a red letter day"is found, paint cell red! }
          Canvas.Brush.Color := clRed;
          Canvas.font.color := clBlack;
          with ARect, Canvas do
            TextRect(ARect, Left + (Right - Left - TextWidth(
              CellValue)) div 2, Top + (Bottom - Top -
              TextHeight(CellValue)) div 2, CellValue);
          break; { leave the loop }
        end; { if }
      end; { for }
    except
      on e : exception do showmessage(inttostr(arow))
    end;
  end; { if }
end;
procedure TCachedCalendar.Click;
const
  BUTTON_LEFT_OFFSET = 10;
  BUTTON_TOP_OFFSET = 10;
var
  Point : TPoint;
  CellValue :  string;
  CellDate : TDatetime;
  SearchDate : TSpecialDateList;
  d : longint;
begin
  GetCursorPos(Point);
  inherited Click;
  CellValue := CellText[Col,Row];
  if ((CellValue<>'') and (Row<>0)) then begin
    try
      CellDate :=
        encodedate(Year, Month, StrToInt(CellValue));
        for d := 0 to (FDateList.count-1) do begin
        SearchDate := FDateList[d];
        if (SearchDate.Date=CellDate) then begin
          if FDatePopupMenu <> nil then FDatePopupMenu.free;
          FDatePopupMenu := TPopupMenu.Create(Self);
          with FDatePopupMenu.Items do begin
            Add(NewItem((FormatDateTime(LongDateFormat,
              SearchDate.Date)), 0, False, true, nil, 0,
              'PopupMenuItem1'));
            Add(NewLine); { Adds a separator bar }
            Add(NewItem(SearchDate.Occasion, 0, False, true,
              nil, 0, 'PopupMenuItem2'));
          end; { with }
          FDatePopupMenu.Popup((Point.x+BUTTON_LEFT_OFFSET),
           (Point.y+BUTTON_TOP_OFFSET));
          break; { leave the loop }
        end; { if }
      end; { for }
    except
      on e : exception do showmessage(inttostr(Row))
    end;
  end; { if }
end;
end.
```