Getting started with Lazarus: Database access

Michaël Van Canneyt

September 2, 2007

Abstract

In the series of articles about Lazarus, the focus was till now on controls. In this article, the focus will shift to what most - if not all - software must do: retrieve and manipulate data. The architecture of data handling in Lazarus will be explained and demonstrated.

1 Introduction

Most business applications display and manipulate data coming from one or more data sources: this data can come from a database server or from a set of files on disk. All these forms of data are accessed using a common database architecture, which is included by default in Lazarus. Many of the standard controls presented in the previous articles come in a data-aware version: they know how to display data coming from this data architecture, and know how to post changes to the data architecture.

The architecture is pure object pascal code: as such it is open, and components exist to access many types of databases. Indeed, Lazarus itself is shipped with components that allow acces to a variety of databases (open source or not) or file formats:

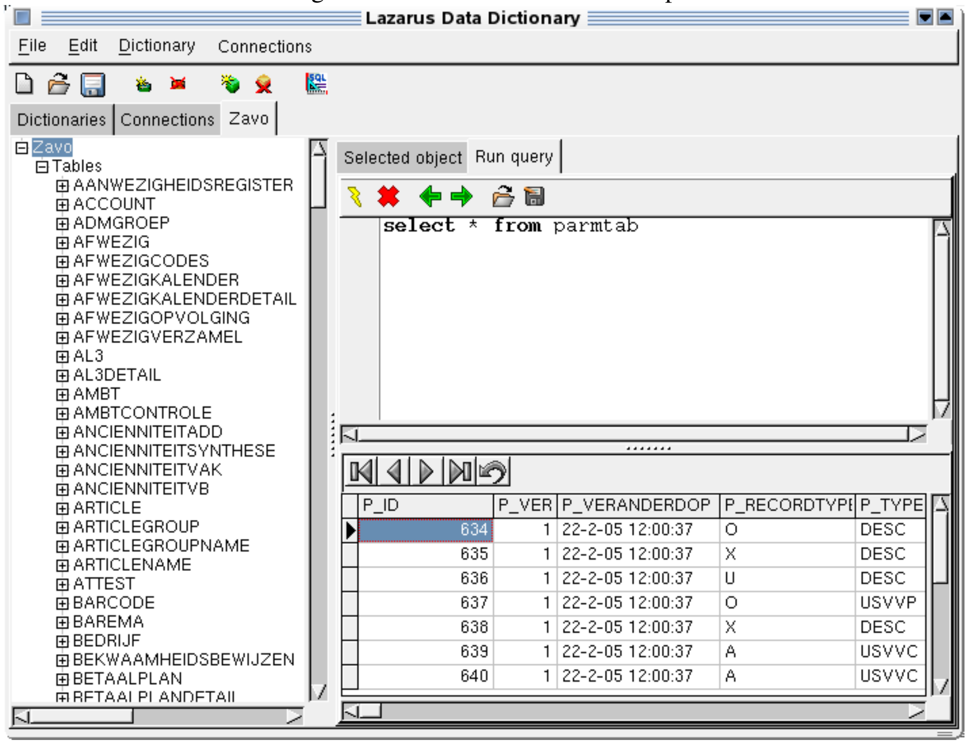
1. Comma-Separated Value (CSV) files.
2. DBF files
3. Firebird or Interbase databases
4. MySQL databases (versions 4.0, 4.1 or 5.0)
5. Oracle databases
6. PostgreSQL databases
7. Sqlite3 embedded databases
8. Any database for which a ODBC driver exists.

There are also components available that can keep in-memory data.

Other than this, there are database access layers provided by third-party developers, such as

1. Advantage Database Server
2. ZeosLib is a set of components that allow to access a variety of databases.

Figure 1: The Lazarus Data Desktop



```
select * from parmtab
```

P_ID	P_VER	P_VERANDERDOP	P_RECORDTYPI	P_TYPE
634	1	22-2-05 12:00:37	O	DESC
635	1	22-2-05 12:00:37	X	DESC
636	1	22-2-05 12:00:37	U	DESC
637	1	22-2-05 12:00:37	O	USVVP
638	1	22-2-05 12:00:37	X	DESC
639	1	22-2-05 12:00:37	A	USVVC
640	1	22-2-05 12:00:37	A	USVVC

Obviously, anyone can write a series of components to acces any other database.

In addition to the components needed to access databases, Lazarus comes with an application called the 'Lazarus Data Desktop'. It can be used to

- Access all lazarus-supported databases.
- Execute SQL queries on the SQL-based databases and view the result.
- Create databases.
- Create common SQL statements for tables.
- Create and maintain data dictionaries, which can be applied to datasets in an application.

A screenshot of the 'Lazarus data desktop' is shown in figure 1 on page 2. A full explanation is outside the scope of this article, and may be left for a future contribution.

2 Architecture

At the heart of the database access architecture lies a set of basic, abstract components:

TDataset As the name suggests, this is the base component for all sets of data. It is a component built around a memory buffer that contains the data - the source of the data is determined by descendents. The data is organized in rows (records), and each row is divided in named columns (fields). It introduces methods to navigate and manipulate the data. It also introduces a cursor in the data: Only the data at the current location of the cursor (the current record) can be shown and manipulated. In order to change another row in the data, the cursor must be moved to this row.

By itself, `TDataset` does not provide access to data: the actual data is fetched by descendent controls, which must override a designated number of abstract, protected methods of `TDataset`, in which they copy the data to the memory buffers managed by `TDataset`, and apply changes to the database or file should the data be changed; However, this is transparent to the user of these descendents.

TField Represents a single field in the current record: It offers methods to retrieve the value of the field. The value can be retrieved as any of the basic types supported by Object Pascal, such as strings, integers, floating point, currency or TDateTime types. In the case of BLOB fields the value can be retrieved as text or in a memory buffer, or it can be saved to file. Various properties exist which dictate how the value should be displayed in a GUI.

TDatasource is the compagnon component of `TDataset`: it communicates changes in the dataset to any third component. In a way, it is the subject of the observer pattern. Many datasources can be connected to a single `TDatasource`. Doing this allows to selectively distribute data events to interested third components, such as GUI controls.

3 Navigating through data

The heart of the data access mechanism is `TDataset`. It introduces all methods, needed to access, navigate and manipulate data. It offers the following important properties:

Active A boolean property. If `True`, the dataset is open, and the data is available for browsing and editing. If `False`, the dataset is closed, and the data is not available. The property can be set to open or close the data. Note that this can cause an exception to be raised, if e.g. the SQL statement contains an error or if the file from which data should be read is not available.

Recno The position of the cursor in the rows of the dataset. This can not always be relied upon: some datasets are unidirectional (only forward navigation is allowed), because the underlying data acces mechanism does not allow navigating back and forth between the rows.

Recordcount is not always available, but indicates the number of records in the set.

BOF The cursor is at the beginning of the set. Moving to the prior record will have no effect.

EOF The cursor is at the end of the set. Moving to the next record will have no effect.

Fields an indexed array of `TField` instances. Each instance represents a field in the current record.

Modified indicates whether the data in the buffer is modified.

State Indicates the current state of the dataset: this is a read-only property. The most used states are `dsInactive` when the dataset is closed, `dsBrowse` when data is being browsed. `dsEdit` is the state when editing the current record, or `dsInsert` when a new record is being added.

FieldDefs a collection of `TFieldDef` items. Each item contains the definition of a field in a row. This collection is filled by `TDataset` descendents so it represents accurately the data one is trying to access. The collection then is used by `TDataset` to create the `TField` instances when the dataset is opened. This happens transparantly, and usually it is not necessary to access this collection - with the notable exception of in-memory datasets.

Filter a filter expression to filter the records in the dataset. Whether this is implemented (and with what syntax) or not depends on the particular `TDataset` engine.

Filtered a boolean property which indicates whether the filter expression in the `Filter` property should be applied or not.

If a dataset is open, and both `EOF` and `BOF` are `True`, this means the dataset is empty (contains no records). It is not advisable to check for `RecordCount=0`, because this may not be implemented for the `TDataset` descendent in use.

The following is the only correct check:

```
Var
  D : TDataset;

begin
  D.Open;
  Try
    if (D.EOF and D.BOF) then
      Raise Exception.Create('No data available!');
    // Do something with D in case there is data.
  Finally
    D.Close;
  end;
end;
```

The following methods are available to navigate the data:

Open opens the dataset: this is equivalent to setting the `Active` property to `True`.

Close closes the dataset: this is equivalent to setting the `Active` property to `False`.

Next moves the cursor to the next record in the dataset. If `EOF` is `True`, this has no effect.

Prior moves the cursor to the previous record in the dataset. If `BOF` is `True`, this has no effect. Not all `TDataset` descendents support this operation.

First moves the cursor to the first record in the dataset. After a call to `First`, `BOF` is `True`. Not all `TDataset` descendents support this operation.

Last moves the cursor to the last record in the dataset. After a call to `Last`, `EOF` is `True`.

MoveTo This method accepts a 1-based record number (`RecNo` as a parameter. It will jump to the indicated record. Not all `TDataset` descendents support this operation.

Refresh refreshes the data in the dataset from it's source.

With these methods and properties, most read-only operations can be performed on a dataset. The following code will for instance walk over all records in the dataset, and perform some operation:

```
Var
  D : TDataset;

begin
  D.Open;
  Try
    While Not D.EOF do
      begin
      DoSomeOperation(D);
      D.Next;
      end;
  Finally
    D.Close;
  end;
end;
```

Note the `try`...`finally` block: if an exception occurs during manipulation of the data, the dataset will be closed anyway at the end of the routine.

4 Accessing data in the fields

To access the field values in the current record, the `Fields` property can be used. This is a property of class `TFields`, which has as a default property an indexed array of `TField` descendent instances. The `TField` class has the following properties:

FieldName the name of the field.

DataType the database native type of the data in this field.

AsInteger the value of the data as an integer.

AsString the value of the data as a string.

AsDateTime the value of the data as a `TDateTime` value.

AsCurrency the value of the data as a `Currency` value.

AsFloat the value of the data as a `Double` value.

AsBoolean the value of the data as a `Boolean` value.

Value the value of the data as a variant.

Each `AsNNN` property will attempt to convert the actual data to the requested type. This means that if the underlying database field is a string field, then reading the value by means of the `AsInteger` property will attempt to convert the string value to an Integer. If the string value in the database is not a valid representation of an integer, then an `EConvertError` will be raised. Conversely, setting the property will convert the value to the type expected by the database.

The following routine dumps all data in a record to the console:

```
Procedure DoSomeOperation(D : TDataset);

Var
  I : Integer;
  F : TField;

begin
  For I:=0 to D.Fields.Count-1 do
    begin
    F:=D.Fields[I];
    Write(F.FieldName:32,' : ');
    If (F.DataType=ftBlob) then
       Writeln('<BLOB>')
    else
       Writeln(F.AsString);
    end;
end;
```

Combined with the navigation example above, this would dump all data in a dataset to screen.

In case the names of the fields are known, the fields can be accessed with the `FieldByName` method of `TFields` or `TDataset`. Continuing the example above, if the Dataset `D` contains the result of an SQL Query like the following:

```
  SELECT FirstName, LastName, BirthDay From Persons;
```

then the `DoSomeOperation` could be coded like this:

```
Procedure DoSomeOperation(D : TDataset);

Var
  BD : TDateTime;

begin
  Writeln('FirstName : ',D.FieldByname('FirstName').AsString;
  Writeln('LastName  : ',D.FieldByname('LastName').AsString;
  BD:=D.Fields.FieldByName('BirthDay').AsDateTime;
  Writeln('BirthDay  : ',DateToStr(BD));
end;
```

Note that `BD` is retrieved from the `Fields` property. `FieldByName` will raise an exception if the dataset does not contain a field with the requested fieldname. If this behaviour is not desirable, the alternate `FindField` method will simply return `Nil` if the requested field is not present. This can be used to check for the existence of a field, like this:

```
Function HasField(D : TDataset; FieldName : String) : Boolean;

begin
  Result:=D.FindField(FieldName)<>Nil;
end;
```

Note that the items in the `Fields` property are not all of type `TField`: they are descendents of `TField`, a descendent exists for each native database type. When fetching data from the database, each `TDataset` will map the native database type for each field in the recordset to a `TField` descendent class, and will set the `DataType` property to a value that describes best the type of the data in the database. Several standard `TField` descendents are available:

TIntegerField for up to 32-bit signed integer values. DataType will be `ftInteger`.

TSmallIntField for up to 16-bit signed integer values. DataType will be `ftSmallInt`.

TWordField for up to 16-bit unsigned integer values. DataType will be `ftWord`.

TLargIntField for 64-bit integer values. DataType will be `ftLargeInt`.

TFloatField for floating point values. DataType will be `fDouble`.

TStringField for string values. DataType will be `ftString`.

TBCDField for BCD values. DataType will be `ftBCD`.

TDateField for date values. DataType will be `ftDate`.

TTimeField for time values. DataType will be `ftTime`.

TDateTimeField for TDateTime values. DataType will be `ftDateTime`.

TBLOBField for BLOB data (binary data without form). DataType will be `ftBLOB`.

TMemoField for BLOB data which contain only text. DataType will be `ftMemo`.

Exactly which field is used depends on the `TDataset` descendant. More specialized `TField` descendents exist, and although they are rarely used, some `TDataset` descendents use specialized `TField` descendents to encapsulate database specific behaviour for the field types.

5 Manipulating data

Till now, all methods and properties were of use for viewing and navigating through data. There are obviously also methods for editing, adding and deleting data:

Edit Puts the dataset in edit mode. After this command, the contents of the current record can be modified. (The `State` property will equal `dsEdit`).

Insert Puts the dataset in insert mode at the current location: a new record is added. After this command, the contents of the new record can be modified. (The `State` property will equal `dsInsert`).

Append Is equal in functionality to `Insert`, only the new record is added at the end of the recordset. The `State` property will equal `dsInsert` as well.

Post If the dataset is in edit or append mode, then the `Post` command will commit the changes to the record buffer, and will put the dataset again in browse mode. Depending on the `TDataset` descendent implementation, the changes will also be committed at once to the underlying database.

Cancel If the dataset is in edit or append mode, then the `Cancel` command will revert any changes and return the dataset to the state it was in prior to the `Edit`, `Insert` or `Append` command.

Delete This command deletes the current record from the record buffer and - depending on the particular `TDataset` being used - from the underlying database. No `Post` is needed to commit this action to the database. It follows that this action is irreversible, and cannot be undone with `Cancel`.

Note that while the dataset is in editing mode, any navigation command will attempt to post the data prior to executing the navigation method itself.

In general, modifying data is then done by putting the dataset in the appropriate editing mode and setting the field properties. The following code could be a `DoSomeOperation` implementation that modifies the data in a dataset:

```
Procedure DoSomeOperation(D : TDataset);

Var
  I : Integer;
  F : TField;

begin
  If D.FieldByName('NeedsCheck').AsBoolean then
    begin
    Edit;
    I:=FieldByName('CheckField').AsInteger;
    FieldByName('CheckField').AsInteger:=I+1;
    FieldByName('NeedsCheck').AsBoolean:=False;
    Post;
    end;
end;
```

The above code will check the boolean field `NeedsCheck`. If it is `True`, it will augment the value of the `CheckField` field with 1, and set the `CheckField` to `False`. In case `DoSomeOperation` is called for all records in a table called `CheckTable`, then the above operation would be equivalent to the following SQL statement:

```
UPDATE CheckTable SET
  CheckField=CheckField+1,
  NeedsCheck=False
WHERE
  NeedsCheck=True;
```

6 Errors when manipulating data

Note that various things can go wrong when editing data in a dataset. Errors are reported through exceptions: the `TDataset` class and it's helper classes do a lot of checking and report any error they find. The usual class used when reporting errors is `EDatabaseError`.

The first error is editing data when the dataset is not in edit mode. The following code will result in a `EDatabaseError` exception:

```
With MyDataset do
  begin
  Open;
  FieldByName('MyField').AsString:='Some string';
  Post;
  end;
```

The `Post` operation will never be reached.

An often encountered error is writing the wrong kind of data to a field. Supposing the `MyField` field is of SQL type `INT` (this will normally correspond to a `TIntegerField` class, and `DataType` value `ftInteger`), then the following code will raise an `EConvertError`:

```
With MyDataset do
  begin
  Open;
  Append;
  FieldByName('MyField').AsString:='Some string';
  Post;
  end;
```

The error is raised because the dataset expects an integer value, and `'Some String'` is not a valid integer.

A third kind of error is not supplying all required fields for a record. In case a dataset has 2 string fields, `MyField` and `MyRequiredField`, then the following will raise an exception when the `Post` command is executed:

```
With MyDataset do
  begin
  Open;
  Append;
  FieldByName('MyField').AsString:='Some string';
  Post;
  end;
```

This will cause an `EDatabaseError` error saying that `MyRequiredField` needs a value.

The same error can also occur when editing data, when a field is cleared:

```
With MyDataset do
  begin
  Open;
  Edit;
  FieldByName('MyField').AsString:='Some string';
  FieldByName('MyRequiredField').Clear;
  Post;
  end;
```

The `Clear` method of `TField` clears the contents of the field - corresponding to the SQL value NULL. It is possible to check if a field is null with the `IsNull` property:

```
Var
  F : TField;

begin
  With MyDataset do
    begin
    Open;
    Edit;
    // Do some other things.
    F:=FieldByName('MyRequiredField');
    If F.IsNull then
      F.AsString:='Some Default';
    Post;
    end;
```

It is important to be aware of the errors that can occur, because the dataset will be left in an inconsistent state after the error occurred, and actions to correct the situation will be required. If an error occurs in the `Post` operation (e.g. when no value for a required field is given), then the dataset will be left in the edit state. The following code for instance will result in an exception:

```
With MyDataset do
  While Not EOF do
    begin
    Try
      Edit;
      FieldByName('MyRequiredField').Clear;
      Post;
    Except
      // Silently ignore any errors.
    end;
    Next;
    end;
```

Even though it looks like the code takes care of errors, it does not: if an exception is raised, then the exception is caught by the `Try..Except` block, but the dataset is left in edit mode. The `Next` statement that follows will try to post the pending changes anyway, and an exception will be raised again - but this time not caught by the `Try..Except` block.

The correct way of handling the error is by canceling the pending changes:

```
With MyDataset do
  While Not EOF do
    begin
    Try
      Edit;
      FieldByName('MyRequiredField').Clear;
      Post;
    Except
      // Silently ignore any errors.
      Cancel;
    end;
    Next;
    end;
```

In case of an error, the `Cancel` will cancel the changes, and put the dataset back in browse state. The `Next` statement will then simply move to the next record.

7 Dataset events

The errors in the previous section make it clear that it is necessary to do some error checking. This is easy when doing everything in linear code as in the examples, but if the data is edited in a GUI and the actions are driven by user-generated events, then error checking is not so easy. Luckily, `TDataset` offers a lot of events to cater for this. For almost all methods discussed in the previous sections, there are 2 events. The events are consistently named:

BeforeXYZ which happens before the method `XYZ` is actually executed. If an exception is raised inside the event handler, the method will not be executed. It can be used to perform checking if all necessary conditions to succesfully execute the method have been fulfilled.

AfterXYZ which happens after the method was actually executed. It can be used to respond on various conditions, such as displaying additional information after a change in the data. If an error occurred during the execution of the method `XYZ`, then this event will be skipped. For instance, `AfterOpen` will not be fired if a SQL-based dataset is opened which has a syntax error in it's SQL statement: the `Open` method will raise an exception, and the `AfterOpen` event will not be called.

The navigation events are an exception to the naming rule: The events related to scrolling through the data are named `AfterScroll` and `BeforeScroll`.

Some of the events will be explained in the following sections.

8 Creating a GUI to display and manipulate data

In order to display data in a GUI, a set of so-called DB-Aware controls exist: these are descendents of standard controls, which communicate with a `TDatasource` instance: They have usually 2 properties:

DataSource the `TDatasource` instance from which data and event notifications should be accepted.

DataField the name of the field in which the control is interested. The control will display and manipulate the value of this field only, for the current record.

Some controls - notably, the Grid and DBNavigator control - do not have a `DataField` property, because they do not act on a single field but on the data as a whole.

In general, the following steps should be taken in order to display and/or manipulate data in a form:

- Drop one or more `TDataset` descendents on the form, and connect them to a database connection component if the descendent requires this. Set the needed properties to select the data one wishes to access (this can be a filename or an SQL query).
- If the datasets will always display the same data, it is good practice to create persistent fields: This defines all fields in the dataset as components, owned by the form. This will allow to set some display properties for the fields, and these display properties will stored and used in the form at runtime. Additionally, the persistent fields provide easy access to the fields: supposing a field `ItemCount` exists in a dataset, then creating persistent fields, and renaming the field component to `ItemCount`, allows to access it's content as follows:

  ```
  Procedur TMyForm.MyMethod;

  begin
    ItemCount.AsInteger:=0;
  end;
  ```

  The field (in the data sense) of the dataset is available as a field (in the Object Pascal sense) of the form class.
- Drop one or more `TDatasource` components on the form, and connect them to the `TDataset` descendent.
- Drop as many DB-Aware controls on the form as needed to edit the data, and connect them to the `TDatasource` components.

If everything is set up correctly, then setting the `Active` property of the `TDataset` instances to `True` will actually display the data in the form while it is being designed in the Lazarus IDE. Most of this can be accomplished without a single line of code.

The following data-aware controls are available by default in the Lazarus IDE:

TDBLabel a simple label control that shows the text of any field.

TDBEdit a simple edit control to edit the contents of a single field.

TDBMemo a simple memo control to edit the contents of a single field as multi-line text. It knows about BLOB fields, and can show them.

TDBImage an image showing control. It shows an image which it can load from a BLOB field.

TDBListBox a listbox control. It allows to select the value of a field from an item in the listbox.

TDBComboBox a combobox control. It allows to select the value of a field from an item in the combobox list.

TDBCheckBox a checkbox control. It allows to show and set a boolean field: the values for the checked and unchecked state can be specified.

TDBRadioGroup a radiogroup control. It allows to show the contents of an integer field, and represents them in the radiogroup by checking the item whose itemindex corresponds to the value of the field.

TDBCalendar a calendar control to show the value of a TDateTime field.

TDBNavigator is a specialized control: it shows a series of buttons. Each button corresponds to a TDataset navigation method, or to one of the data manipulation methods. Clicking the appropriate button will execute the corresponding action on the dataset. The control keeps track of the state of the dataset, and disables the buttons that cannot be used in the current state of the dataset.

TDBGrid shows the contents of the dataset in a grid: not only the current record is shown, but all records that fit in the grid are shown. The data in the grid is editable: clicking any cell will edit the data in the cell.

When compared to their non-data aware ancestors, most of these controls do not have additional properties except the `Datasource` and `Datafield` properties. The components will be demonstrated in some simple applications further on.

9 The simplest dataset: an In-Memory dataset

Till now, only abstract ideas have been discussed. In the subsequent, some actual implementations of `TDataset` will be discussed, and the most simple example is an in-memory dataset.

The LazMemds package registers in the Lazarus IDE the `TMemDataset` dataset: this is a Memory Dataset which keeps any data that is written to it in memory. It has methods to load and save the data to disk, and a method to define a new dataset. In real applications, it's not really recommended to use `TMemDataset` for anything else than lookup lists, but it can nevertheless be used as a persistent data store. The ability to store data on disk will be used to write a small address book application, which will demonstrate how to create a dataset in memory or load one from disk.

To do this, a new project is created, with a single form `TMainForm`. On the form, a menu is created with 4 items:

New to create a new address book.

Save to save an address book to file.

Open to load an address book from file.

Quit to quit the application.

After this, a `TMemDataset` component is dropped on the form (named`MDAdresses`), together with a `TDatasource` component (`DSAddresses`), and the `Dataset` property of the `TDatasource` is set to the memdataset component. All is now in place for the form to start showing data.

In the `OnClick` event handles of the `New` menu, the following code is created:

```
procedure TMainForm.MINewClick(Sender: TObject);
begin
  NewBook;
end;

procedure TMainForm.NewBook;

begin
  With MDAddresses do
    begin
    Close;
    FieldDefs.Clear;
    FieldDefs.Add('FirstName',ftString,50);
    FieldDefs.Add('LastName',ftString,50);
    FieldDefs.Add('Email',ftString,100);
    FieldDefs.Add('Mobile',ftString,20);
    FieldDefs.Add('Private',ftBoolean,20);
    FieldDefs.Add('Comment',ftString,1024);
    CreateTable;
    FileName:='';
    Open;
    end;
end;
```

The `NewBook` procedure first closes the dataset. If the dataset was already closed, this action does nothing, so it is safe to call. After that the `FieldDefs` collection is cleared, and definitions for the fields needed in the application are added to it. Based on these definitions, the `CreateTable` method of `TMemDataset` will then create the internal structures needed to maintain data corresponding to the field definitions. Finally, the `FileName` property is cleared, and the dataset is opened. The filename must be cleared, because if it is set, the `Open` method will attempt to locate the file and load the data from the file - overwriting the newly-created structures.

The `OnClick` handler of the `Open` menu item does just this:

```
Procedure TMainForm.MIOpenClick(Sender: TObject);
begin
  With ODAddr do
```

```
    If Execute then
      LoadFromFile(FileName);
end;

procedure TMainForm.LoadFromFile(AFileName : String);

begin
  With MDAddresses do
    begin
    LoadFromFile(AFileName);
    FileName:=AFileName;
    Open;
    end;
end;
```

The `ODAddr` component is a `TOpenDialog`, and will present the user with a dialog to choose a file. If the user chose a file, then the `LoadFromFile` method of the form is called, and this will simply call the `LoadFromFile` method of `TMemDataset`. This method loads the fielddefs from the file as well as the data to be maintained in memory, but it does not open the dataset. Only after `Open` was called, the dataset will be opened and the data will be available for editing.

The `Save` menuitem will do something similar:

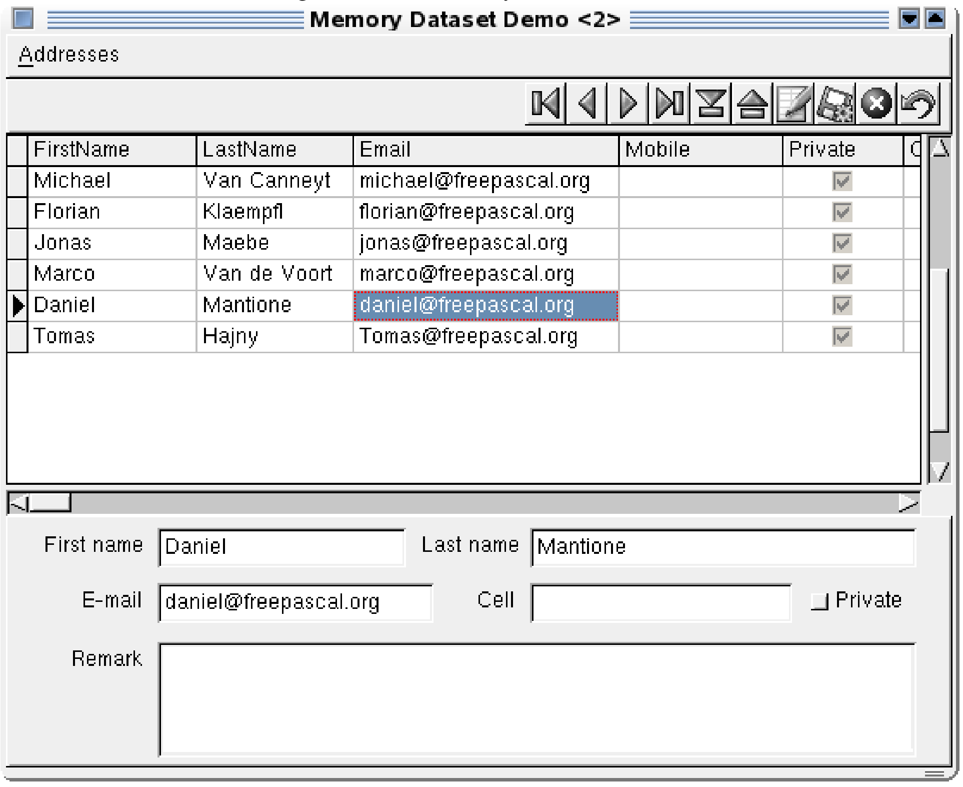
```
procedure TMainForm.MISaveClick(Sender: TObject);
begin
  With SDAddr do
    if Execute then
      SaveToFile(FileName);
end;

procedure TMainForm.SaveToFile(AFileName : String);

begin
  With MDAddresses do
    begin
    SaveToFile(AFileName);
    FileName:=AFileName;
    end;
end;
```

Setting the `FileName` property ensures that the data is saved to the filename when the dataset is closed.

Now all is ready to actually display and edit the data. To display the data, a top-aligned panel with a `TDBNavigator` component is dropped on the form, and the `DataSource` property of the navigator is set to the `DSAddresses` datasource. A second panel is dropped, (bottom aligned) and on it, 4 `TDBedit` controls, a `TDBCheckBox` control and a `TDBMemo` control are dropped. Like for the navigatore control, the `Datasource` property of all these edits is set to the `DSAddresses` datasource. The `DataField` property of the 4 edit controls are set to the fieldnames `FirstName`, `LastName`, `EMail` and `Mobile`. The checkbox is set to point to the `Private` field, and the `TDBMemo` will contain the `Comment` field.

Last but not least, a TDBGrid is dropped between the 2 panels, and it's alignment set to `alClient` so it fills the space between the 2 panels. Its `Datasource` property is set to

Figure 2: The memory dataset at work



	FirstName	LastName	Email	Mobile	Private	C
	Michael	Van Canneyt	michael@freepascal.org		✓	
	Florian	Klaempfl	florian@freepascal.org		✓	
	Jonas	Maebe	jonas@freepascal.org		✓	
	Marco	Van de Voort	marco@freepascal.org		✓	
▶	Daniel	Mantione	daniel@freepascal.org		✓	
	Tomas	Hajny	Tomas@freepascal.org		✓	

First name: Daniel Last name: Mantione

E-mail: daniel@freepascal.org Cell: ☐ Private

Remark:

`DSAddresses`.

Now the application is completely ready to go. Some more cosmetic changes (adding labels, setting alignments) may be applied. When running the finished application, after adding some data, it will look something like figure 2 on page 15.

10 Adding some structure: CSV data

A common format to transport data is in a CSV format: Comma Separated Values. Lazarus offers a `TSDFDataset` descendent of `TDataset` to deal with this. It is contained in the `LazSDF` package.

This `TDataset` descendent does not differ a lot from `TMemDataset` in it's use. It has 3 important properties:

Delimiter The character used to delimit records in a line. By default this is the comma (,).

FirstLineAsSchema This tells `TSDFDataset` to treat the first line as a line containing the fieldnames.

FileName The name of the file to read when 'Open' is called.

To demonstrate the use of this component, a small application will be created that can be used as a CSV viewer.

It's beginnings are much the same as the memdataset demo application:

- A `TSDFDataset` instance (named `CSV`).

- A main menu with New, Open, Save and 'Save as' menu items under the file menu.
- A panel with a checkbox (CBFirstLineAsSchema) and a DB navigator control.
- A grid, which fills the whole form.
- `TOpenDialog` and `TSaveDialog` components, named `ODCSV` and `SDCSV`.

The menu event handlers look much the same as in the previous application:

```
procedure TMainForm.SaveToFile(ForceRename : Boolean);

begin
  If ForceRename or (CSV.FileName='') then
    With SDCSV do
      If Not Execute then
        Exit
      else
        CSV.FileName:=FileName;
  CSV.SaveFileAs(CSV.FileName);
end;
```

The `ForceRename` parameter will be set to `True` if the method is called from the `Save As` menu item. If so, the `SDCSV` dialog will be used to get a value for the `Filename` property of the CSV component. After this the `SaveFileAs` method of the `CSV` component is used to actually save the data.

To load a file is slightly more complicated:

```
procedure TMainForm.LoadFromFile(AFileName : string);

Var
  B : Boolean;
  C : Char;

begin
  C:=DetermineSeparator(AFileName,B);
  If (C=#0) then
    exit;
  With CSV do
    begin
    Delimiter:=C;
    FirstLineAsSchema:=B;
    CBFirstLineAsSchema.Checked:=B;
    FileName:=AFileName;
    Open;
    end;
end;
```

The first thing to do is take an educated guess at the values for the `Delimiter` and `FirstLineAsSchema` properties. This is done in the `DetermineSeparator` routine. After the values have been obtained, the properties are set, as well as the `FileName` property. Finally, the dataset is opened.

The `DetermineSeparator` routine takes a straightforward approach to guessing the delimiter character:

```
Function TMainForm.DetermineSeparator(AFileName : string;
                          var HasFieldNames : boolean) : Char;

Const
  Seps : Array[1..5] of char = (',',';',#9,'@','#');

Var
  F : TextFile;
  S,S2,T : String;
  I : Integer;

begin
  AssignFile(F,AFileName);
  Reset(F);
  Try
    ReadLn(F,S);
  Finally
    CloseFile(F);
  end;
  Result:=#0;
  I:=0;
  While (Result=#0) and (I<5) do
    begin
    Inc(I);
    If (Pos(Seps[i],S)<>0) then
      Result:=Seps[i]
    end;
  If (Result=#0) then
    begin
    S2:=Format(SDelimiterChar,[Copy(S,1,40)]);
    If InputQuery(SDelimiter,S2,T) then
      if Length(T)=1 then
        Result:=T[1]
      else
        begin
        T:=Format(SErrInvalidSeparator,[T,Length(T)]);
        MessageDLG(T,mtError,[mbOK],0);
        end;
    end;
  If (Result<>#0) then
    HasFieldNames:=(Pos(' ',S)=0) and (Pos(Result+Result,S)=0);
end;
```

It simply reads the first line of the file, and then tests for the presence of some well-known separator characters. If none is found, the user is prompted for a character. After this, the `HasFieldNames` parameter is determined by checking whether the first line contains no spaces or empty fields.

If the routine didn't guess the `FirstLineAsSchema` parameter correct, it can be set by the user by checking or unchecking the checkbox at the top of the main window. In it's `OnChange` handler, the following code is executed:

```
procedure TMainForm.CBFirstLineAsSchemaChange(Sender: TObject);

Var
```

```
  B : Boolean;

begin
  With CSV do
    begin
    B:=Active;
    Close;
    FirstLineAsSchema:=CBFirstLineAsSchema.Checked;
    If B then
      Open;
    end;
end;
```

Note that the dataset is closed prior to setting the `FirstLineAsSchema` property. Setting this property while the dataset is open will result in an exception. If the dataset was open to start with, it is reopened again after setting the property.

The `TDBGrid` grid in which the data is shown has quite some options controlling it's appearance and behaviour. The most important properties are:

AutoFillColumns Will resize the columns so they fit the grid.

AutoEdit will automatically set the dataset in edit mode if the user starts

Options a set property with a lot of visual options;

TitleStyle The style of the title columns.

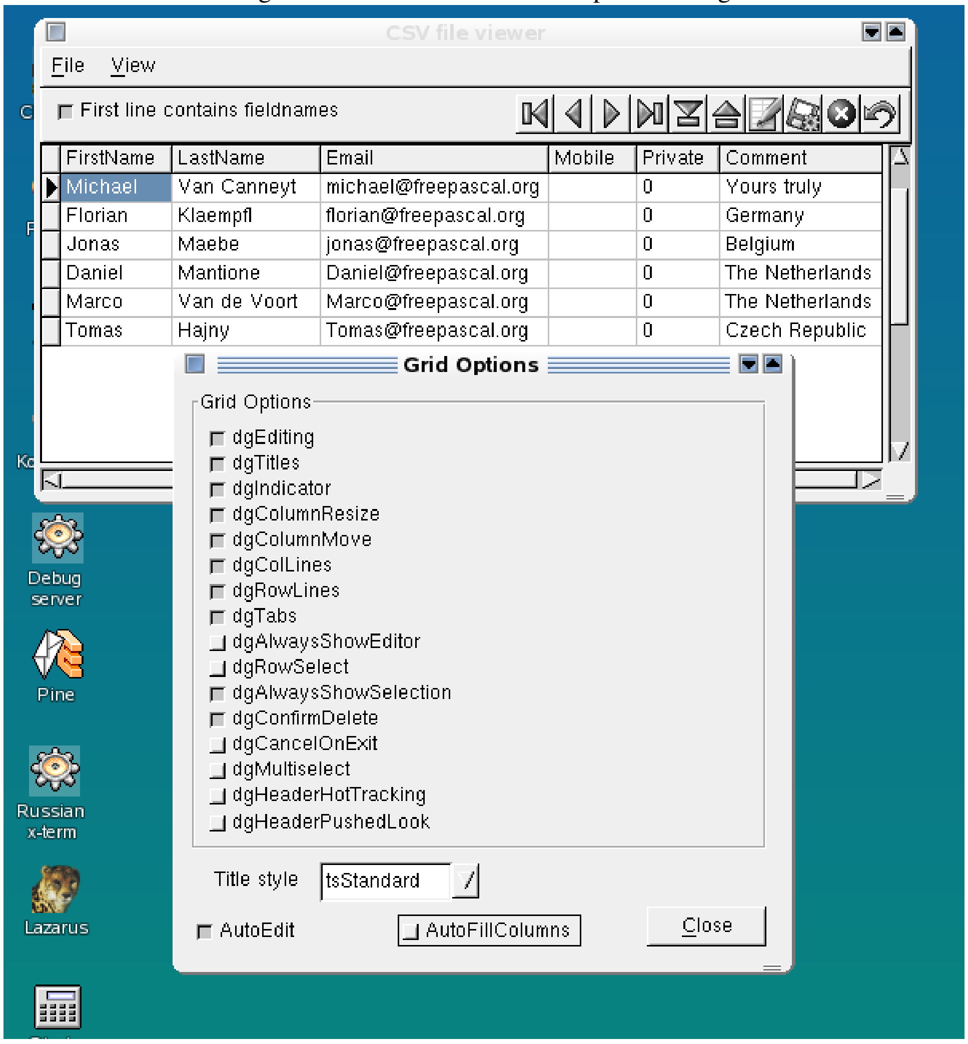
To demonstrate the effect of these options, a small second form is created which can be shown alongside. The RTTI controls on this form are linked to the properties of the `TDBGrid`. The `View|Grid options` menu item can be used to show this form. When shown, it looks like figure figure 3 on page 19. The effect of the various elements in the Options property can be examined by setting them in the options form. Obviously, the `TSDFDataset` component can also be used to create a new CSV file. This is done quite simple in the `OnClick` event handler of the 'New' menu item:

```
procedure TMainForm.MINewClick(Sender: TObject);
begin
  With SDCSV do
    If Execute then
      NewCSV(FileName);
end;
```

To create a new file, a filename is needed, so the handler starts by asking for a filename. If the user didn't cancel, then the `NewCSV` method is called, passing the filename as a parameter:

```
procedure TMainForm.NewCSV(AFileName  : String);

begin
  With TNewCSVForm.Create(Self) do
    Try
      If (ShowModal=mrOK) then
        begin
        CSV.Close;
        CSV.Schema.Assign(Schema);
```

Figure 3: The CSV viewer with options dialog



☑ First line contains fieldnames

FirstName	LastName	Email	Mobile	Private	Comment
Michael	Van Canneyt	michael@freepascal.org		0	Yours truly
Florian	Klaempfl	florian@freepascal.org		0	Germany
Jonas	Maebe	jonas@freepascal.org		0	Belgium
Daniel	Mantione	Daniel@freepascal.org		0	The Netherlands
Marco	Van de Voort	Marco@freepascal.org		0	The Netherlands
Tomas	Hajny	Tomas@freepascal.org		0	Czech Republic

Grid Options

- ☑ dgEditing
- ☑ dgTitles
- ☑ dgIndicator
- ☑ dgColumnResize
- ☑ dgColumnMove
- ☑ dgColLines
- ☑ dgRowLines
- ☑ dgTabs
- ☐ dgAlwaysShowEditor
- ☐ dgRowSelect
- ☑ dgAlwaysShowSelection
- ☑ dgConfirmDelete
- ☐ dgCancelOnExit
- ☐ dgMultiselect
- ☐ dgHeaderHotTracking
- ☐ dgHeaderPushedLook

Title style: tsStandard

☑ AutoEdit ☐ AutoFillColumns

```
      CSV.FirstlineAsSchema:=False;
      CSV.FileName:=AFileName;
      CSV.FileMustExist:=False;
      CSV.Open;
      end;
  Finally
    Free;
  end;
end;
```

This method uses a small auxiliary form, which has a `Schema` property of type `TStrings`. Each string item in the stringlist will be treated as a fieldname for the dataset. After assigning the stringlist to the `Schema` property of the `TSDFDataset`, the `FileName` and `FirstLineAsSchema` must be set. To create a new dataset, the `FileMustExist` property should be set to `False`, and then `Open` should be called to actually create and open the dataset.

11 Conclusion

In this article, the architecture of the data access mechanism in Free Pascal/Lazarus has been presented, and demonstrated with 2 specific examples that operate on data in memory or in simple flat files with comma-separated values. In addition to `comma-delimited` files, there is also a `TFixedFormatDataSet` component available. It works exactly the same as the `TSDFDataset`, except that it expects a file with fixed-length data instead of comma-separated data.

In the next and last article, the components for accessing SQL-Based databases and DBF files will be examined in more detail. They offer more possibilities than the simple components presented here.