The Lazarus Data Desktop

Michaël Van Canneyt

December 4, 2007

Abstract

Lazarus comes with a lot of components to connect to various databases and create data-aware forms. It also comes with a tool which aids in developing database applications: the lazarus data desktop. In this article, the design goals and possibilities of this Lazarus compagnon are explained.

1 Introduction

Many - if not most - applications connect somehow to a database. For this reason, Lazarus comes with a large set of data-aware visual components, and with a set of components to connect to many different databases. A lot of other database connectivity components exist. Additionally, object persistence frameworks such as InstantObjects and tiOPF provide additional ways of creating database applications, using an object oriented paradigma.

Till recently, extra tools were needed to actually manage the databases or test queries on SQL-based databases. Additionally, no data dictionary existed: when creating many forms that often access the same data in different ways, one ends up repeating the same tasks: setting displaynames, field sizes, hints for controls and so on: For this a structure is needed which contains the values of these properties for all tables and fields in a database. This is he data dictionary. The data dictionary can - if so desired - be used to re-create or remodel the database.

The Lazarus Data Desktop is designed to fill this gap: it is a database access tool and a data dictionary manager. Additionally, it allows to create Object Pascal code to manage and access the databases, and is integrated with the Lazarus IDE: A data dictionary created in the Data Desktop can be accessed from within the IDE and applied to a TDataset descendent.

Currently, it features the following possibilities:

- Access to many databases supported by FPC: DBF files, Mysql, Firebird, Oracle, PostGreSQL, SQLite and any ODBC supported database.
- Create a data dictionary for a database, where all the `TField` properties can be stored.
- The data dictionary can be stored in .ini files, or in a database.
- Create DDL and DML SQL statements from the data dictionary.
- Examine table data
- Execute any query, and examine the result.
- Export data to many formats, e.g. XML, JSON, RTF, Plain text (CSV), DBF, SQL.

- Create code from datasets: create an object corresponding to the data in the result of a query, and code to load the object's properties from the query. Creating TiOPF code is supported as well.

All of this functionality is present in the Free Pascal code base, and most of it is available in the Lazarus IDE itself, on the component palette. The database desktop just exposes this functionality in a tool that is designed to handle everyday database programming tasks.

The API managing all these tasks is designed to be extensible: They can be extended to suit the needs of the project at hand. Depending on what needs to be done, a new class should be written and registered, or an existing class can be enhanced: Since the sources of the database desktop are shipped with Lazarus, it can be recompiled with all the desired functionality.

2 Getting started

The Lazarus Data Desktop is under continuous development, as are the APIs that it uses. The best one can do is take a recent version from SVN and recompile it. It is located in the

```
tools/lazdatadesktop
```

directory. To be able to compile it, several lazarus packages are needed:

lazDBF for DBF support.

lazSQLDB for SQLDB support.

lazDataDict for the data dictionary integration in the IDE.

lazDBExport for the export components.

RuntimeTypeInfoControls for RTTI controls. The APIs and the Data Dictionary editor make extensive use of this package.

These packages need not be installed in the IDE, but doing so makes sure that the necessary unit paths are added to the compiler path when compiling.

Opening the lazdatadesktop project and compiling it should result in a working binary.

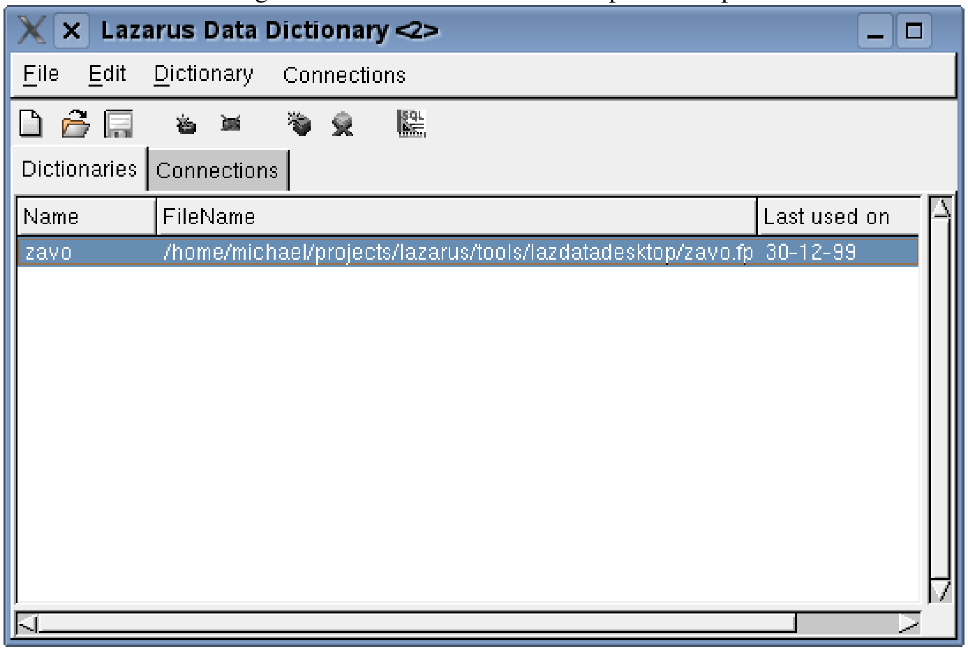
3 First steps

If the data desktop is started, a form as in figure 1 on page 3 should appear. On startup, a list of recently used data dictionaries (on the first tab) or database connections (the second tab) is shown. Double-clicking an entry will open it in a new tab.

Connections can be used to connect to databases: The structure and data of the tables in the database can be viewed. If the database is SQL based, then query statements can be executed on the database. The results of the queries can be viewed, and possibly exported.

Data dictionaries are repositories with meta-information about databases: a data dictionary can be created from zero, but can also be reverse-engineered from an existing database. A data dictionary can be accessed from within a Lazarus program to set field and form properties at run-time (for instance for ad-hoc queries), but can also be applied in the Lazarus IDE at design time.

Figure 1: The Lazarus Data Desktop on startup



Name	FileName	Last used on
zavo	/home/michael/projects/lazarus/tools/lazdatadesktop/zavo.fp	30-12-99

4 Data dictionaries

To create a new data dictionary, the 'File|New' menu can be used. This will start a new data dictionary. Alternatively, the 'Dictionary|Import' menu can be used to reverse engineer a database and create a new, or update an existing, data dictionary. For the purpose of this article, the DBF file with the contents of all Toolbox articles will be imported, by selecting the inhalt.dbf file. This should result in a new data dictionary, which will be saved as - of course - toolbox. After expanding the tree with the data structures, a list of field definitions is visible. Clicking on one of the fields will display the properties for the field in the right half of the screen: these can be edited. The main form should look more or less as in figure 2 on page 4. Obviously, entering the information for all the fields is a lot of work. The advantage of the data dictionary is that this works needs to be done only once.

Once the data dictionary is complete, 2 things can be done:

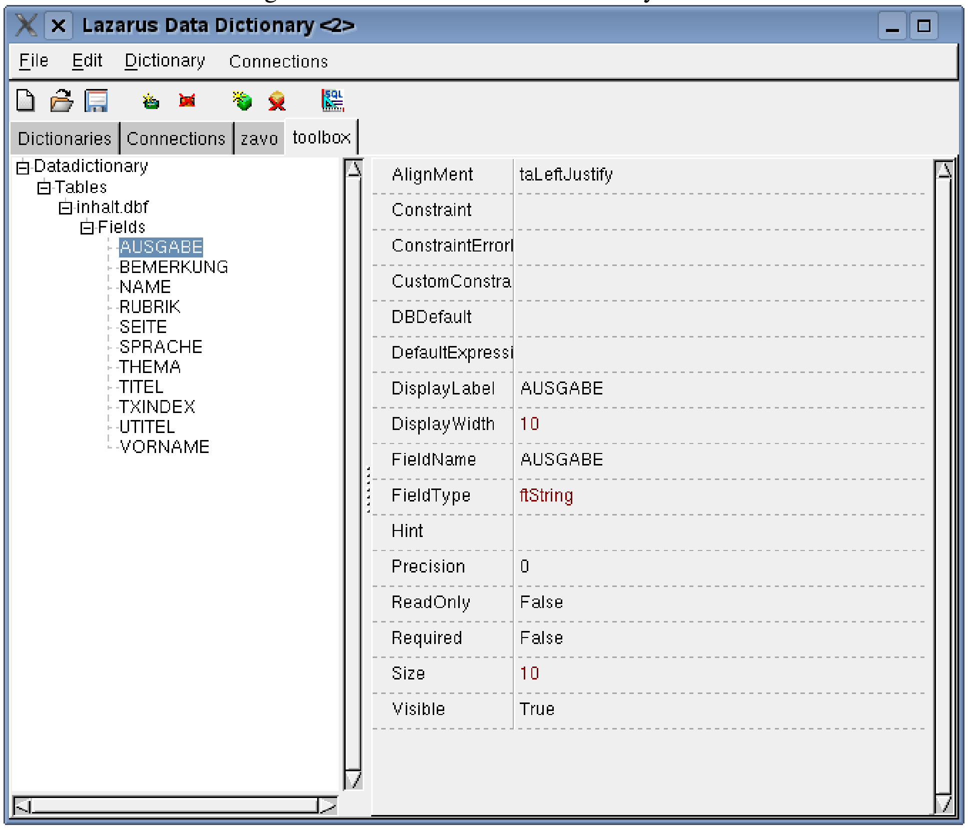
1. Use it in a Lazarus application (designtime or runtime)
2. Generate SQL statements.

The latter can be done with the 'Generate SQL' button or menu item. When clicked, it will pop up a dialog, which looks like a much-extended version of the UpdateSQL component editor found in Borland Delphi. This dialog allows to create SQL statements for selecting, updating, deleting, inserting data from a table in the data dictionary. It provides some options to determine the form of the SQL statements. Additionally, it allows to create a SQL statement which will re-create the table (or part of it) in SQL.

The dialog looks like figure figure 3 on page 4, with some fields selected. Generating the `Inhalt` table from toolbox in a SQL-based database could be done with the following SQL statement:

```
CREATE TABLE inhalt (
  AUSGABE VARCHAR(10),
```

Figure 2: The Lazarus Data Dictionary editor



- Datadictionary
  - Tables
    - inhalt.dbf
      - Fields
        - AUSGABE
        - BEMERKUNG
        - NAME
        - RUBRIK
        - SEITE
        - SPRACHE
        - THEMA
        - TITEL
        - TXINDEX
        - UTITEL
        - VORNAME

AlignMent	taLeftJustify
Constraint	
ConstraintErrorI	
CustomConstra	
DBDefault	
DefaultExpressi	
DisplayLabel	AUSGABE
DisplayWidth	10
FieldName	AUSGABE
FieldType	ftString
Hint	
Precision	0
ReadOnly	False
Required	False
Size	10
Visible	True

Figure 3: Generating SQL from the Data Dictionary

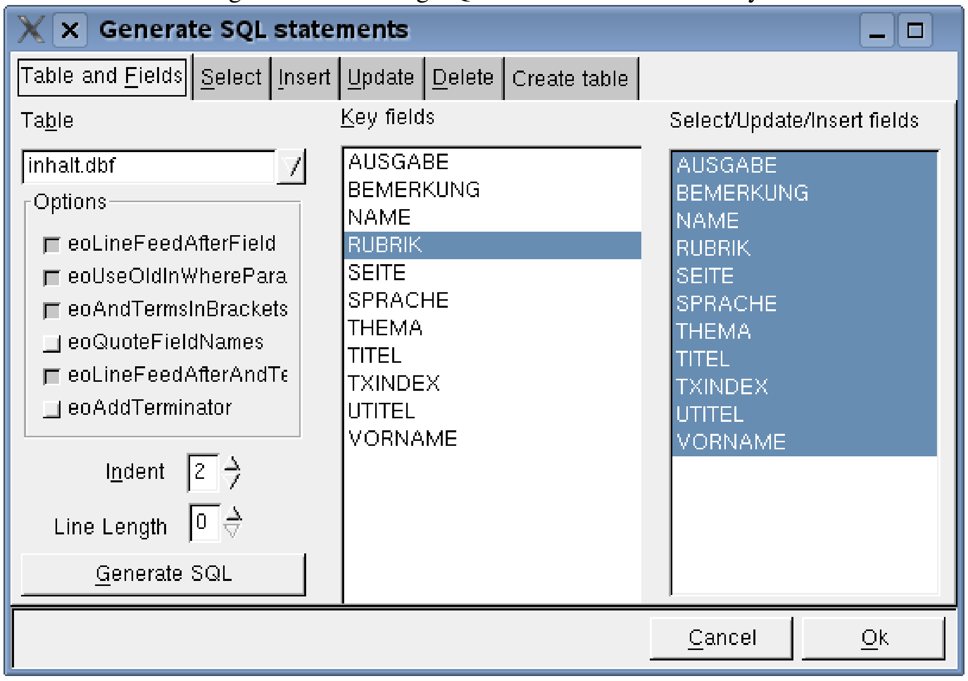


Table: inhalt.dbf

Options
- ☑ eoLineFeedAfterField
- ☑ eoUseOldInWherePara
- ☑ eoAndTermsInBrackets
- ☐ eoQuoteFieldNames
- ☑ eoLineFeedAfterAndTe
- ☐ eoAddTerminator

Indent 2

Line Length 0

Key fields	Select/Update/Insert fields
AUSGABE	AUSGABE
BEMERKUNG	BEMERKUNG
NAME	NAME
RUBRIK	RUBRIK
SEITE	SEITE
SPRACHE	SPRACHE
THEMA	THEMA
TITEL	TITEL
TXINDEX	TXINDEX
UTITEL	UTITEL
VORNAME	VORNAME

```
  BEMERKUNG VARCHAR(74),
  NAME VARCHAR(30),
  RUBRIK VARCHAR(30),
  SEITE VARCHAR(3),
  SPRACHE VARCHAR(30),
  THEMA VARCHAR(30),
  TITEL VARCHAR(74),
  TXINDEX VARCHAR(14),
  UTITEL VARCHAR(74),
  VORNAME VARCHAR(30),
  CONSTRAINT inhalt_PK PRIMARY KEY (AUSGABE))
```

To use the data dictionary during run-time in a Lazarus application, the following code can be used:

```
uses fpdatadict,dbf,db;

var
  DD : TFPDataDictionary;
  DBF : TDBF;
begin
  DD:=TFPDataDictionary.Create;
  DD.LoadFromFile('toolbox.fpd');
  DBF:=TDBF.Create(Nil);
  DBF.TableName:='inhalt.dbf';
  DBF.Open;
  DD.Tables.TableByName('inhalt.dbf').ApplyToDataset(DBF);
  // Alternatively
  // DD.ApplyToDataset(DBF);
end.
```

This code will create a data dictionary, and load it from file. After this, it will load the DBF file, and apply the data dictionary to it. As a result, TField properties such as DisplayName, DisplaySize will be filled in: if the data is displayed in a grid, the proper column sizes and titles will be shown.

5 IDE support for data dictionaries

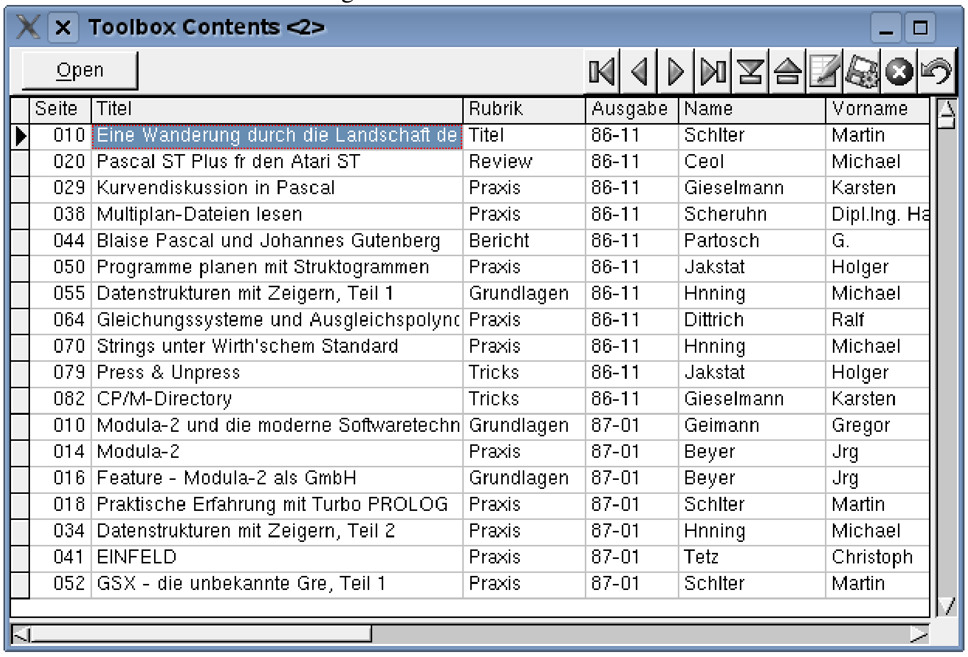
Instead of applying the data dictionary at run-time, the same can be done in the lazarus IDE during design time: if the `lazDataDict` package is installed, then 3 things will be inserted in the IDE:

1. A `Data dictionary` menu item is registered under the 'Project' menu: it allows to select and configure the data dictionary for a project: the name of the data dictionary will be saved with the project options, and the data dictionary will be reloaded when the project is loaded the next time.

2. A menu item 'Database Desktop' is created under the 'Tools' menu: it starts the Lazarus Data desktop. When used the first time, it may ask where the lazarus database desktop application is located.

3. In the form designer, a 'Data dictionary' menu item is registered in the context popup menu: this can be used to apply the data dictionary to any `TDataset` descendent.

Figure 4: The toolbox contents



	Seite	Titel	Rubrik	Ausgabe	Name	Vorname
▶	010	Eine Wanderung durch die Landschaft de	Titel	86-11	Schlter	Martin
	020	Pascal ST Plus fr den Atari ST	Review	86-11	Ceol	Michael
	029	Kurvendiskussion in Pascal	Praxis	86-11	Gieselmann	Karsten
	038	Multiplan-Dateien lesen	Praxis	86-11	Scheruhn	Dipl.Ing. Ha
	044	Blaise Pascal und Johannes Gutenberg	Bericht	86-11	Partosch	G.
	050	Programme planen mit Struktogrammen	Praxis	86-11	Jakstat	Holger
	055	Datenstrukturen mit Zeigern, Teil 1	Grundlagen	86-11	Hnning	Michael
	064	Gleichungssysteme und Ausgleichspolync	Praxis	86-11	Dittrich	Ralf
	070	Strings unter Wirth'schem Standard	Praxis	86-11	Hnning	Michael
	079	Press & Unpress	Tricks	86-11	Jakstat	Holger
	082	CP/M-Directory	Tricks	86-11	Gieselmann	Karsten
	010	Modula-2 und die moderne Softwaretechn	Grundlagen	87-01	Geimann	Gregor
	014	Modula-2	Praxis	87-01	Beyer	Jrg
	016	Feature - Modula-2 als GmbH	Grundlagen	87-01	Beyer	Jrg
	018	Praktische Erfahrung mit Turbo PROLOG	Praxis	87-01	Schlter	Martin
	034	Datenstrukturen mit Zeigern, Teil 2	Praxis	87-01	Hnning	Michael
	041	EINFELD	Praxis	87-01	Tetz	Christoph
	052	GSX - die unbekannte Gre, Teil 1	Praxis	87-01	Schlter	Martin

Obviously, this will only work if persistent fields have been created for the dataset. (the fields editor should be used for this). For this to work, a data dictionary must have been selected for the current project.

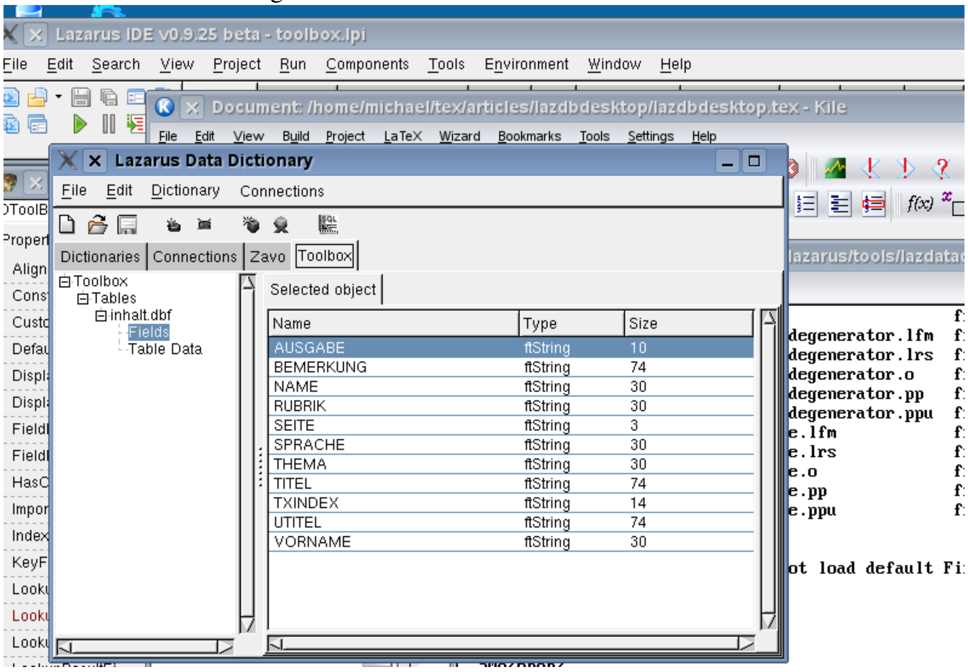
To illustrate all this, the just created 'Toolbox' data dictionary can be applied to a small program: The program is extremely simple: a TDBF component with tablename 'inhalt.dbf', a `TDBGrid` and a `TDBNavigator` component. A Button with the following `OnClick` method:

```
Procedure TToolBoxForm.BOpenClick(Sender: TObject);

Var
  DD : TFPDataDictionary;

begin
  DD:=TFPDataDictionary.Create;
  Try
    DD.LoadFromFile('toolbox.fpd');
    DD.Tables.TableByName('Inhalt.dbf').ApplyToDataset(DToolbox);
    DToolbox.open;
  Finally
    DD.Free;
  end;
end;
```

Running the program and clicking the `Open` button should result in something like figure 4 on page 6 Note that the titles of the grid show not the all-capitals fieldname, but the displayname as set in the data dictionary. Fields marked as 'Invisible' are not shown.

The same effect can be achieved at runtime by choosing the 'toolbox' data dictionary in the 'Project|Data Dictionary|Set...' menu. After this, persistent fields should be created (using

Figure 5: Connected to the toolbox database


Lazarus IDE v0.9.25 beta - toolbox.lpi
File Edit Search View Project Run Components Tools Environment Window Help
Document: /home/michael/tex/articles/lazdbdesktop/lazdbdesktop.tex - Kile
Lazarus Data Dictionary
File Edit Dictionary Connections
Dictionaries Connections Zavo Toolbox
Toolbox
Tables
inhalt.dbf
Fields
Table Data
Selected object

Name	Type	Size
AUSGABE	ftString	10
BEMERKUNG	ftString	74
NAME	ftString	30
RUBRIK	ftString	30
SEITE	ftString	3
SPRACHE	ftString	30
THEMA	ftString	30
TITEL	ftString	74
TXINDEX	ftString	14
UTITEL	ftString	74
VORNAME	ftString	30

the standard 'Fields editor') and the data dictionary can be applied. The IDE messages window will show a message that the data dictionary was applied, and what fields in the dataset didn't have a corresponding field in the data dictionary. The effect can be seen by selecting and inspecting a field's properties in the object inspector. Note that the code above in the `OpenClick` is no longer necessary when this is done.

6 Connecting to databases

The second functionality of the data desktop is simply connecting to existing databases and examine the data in it's tables or running queries on it (in the case of an SQL-enabled database). All databases supported by FPC can be connected to: All that is needed is a small class that registers the database support in the data dictionary engine, and the database desktop will pick it up and offer a connection type.

Connecting to a database is done through the 'Connection|New' menu item. Below this item, a list of supported connection types is shown. After a type was chosen, the connection parameters should be given, and when the connection was succesfully established, a new tab will be shown with the contents of the database: currently, only tables will be shown.

To connect to a 'database' with DBF files, the directory with the DBF files must be entered. So, to connect to the toolbox database, a new 'DBF Files' connection is started, the path to the DBF file is chosen, and the connection is saved with the 'Toolbox' name. When done, the screen should look like figure 5 on page 7. At the right of the window is a work area. It can show the definition of the object chosen at the left, or it can - in the case of a table - show the contents of the table - a view much like the one in figure 4 on page 6. The data of a table can be edited, if one should wish to do so: creating test data for an application is one of the uses of the Data Desktop.

However, much more is possible. The 'Data' view has - beside the navigation buttons - 2 extra buttons that allow to

1. Item export the data to a variety of formats. Currently, 8 formats are available.
2. Create various kinds of Object Pascal code based on the shown data. Currently, 4 kinds of code can be generated.

7 Exporting data

Exporting the shown data can be done to the following formats:

1. CSV (comma separated values) text file.
2. Fixed-length text file.
3. A simple XML file.
4. A simple JSON file.
5. SQL statements to fill a table with the data shown.
6. A LaTeX table.
7. A RTF table.
8. DBF files.

The above is the list of exporters currently implemented in Free Pascal (more are planned). As can be seen, this functionality is suitable for transforming data from one data format to another. The list of export mechanisms can be extended easily with self-written exporters: the API for this is in the Free Pascal sources. The relevant unit is called fpdbexport, and the many available examples can be studied to write custom export mechanisms.

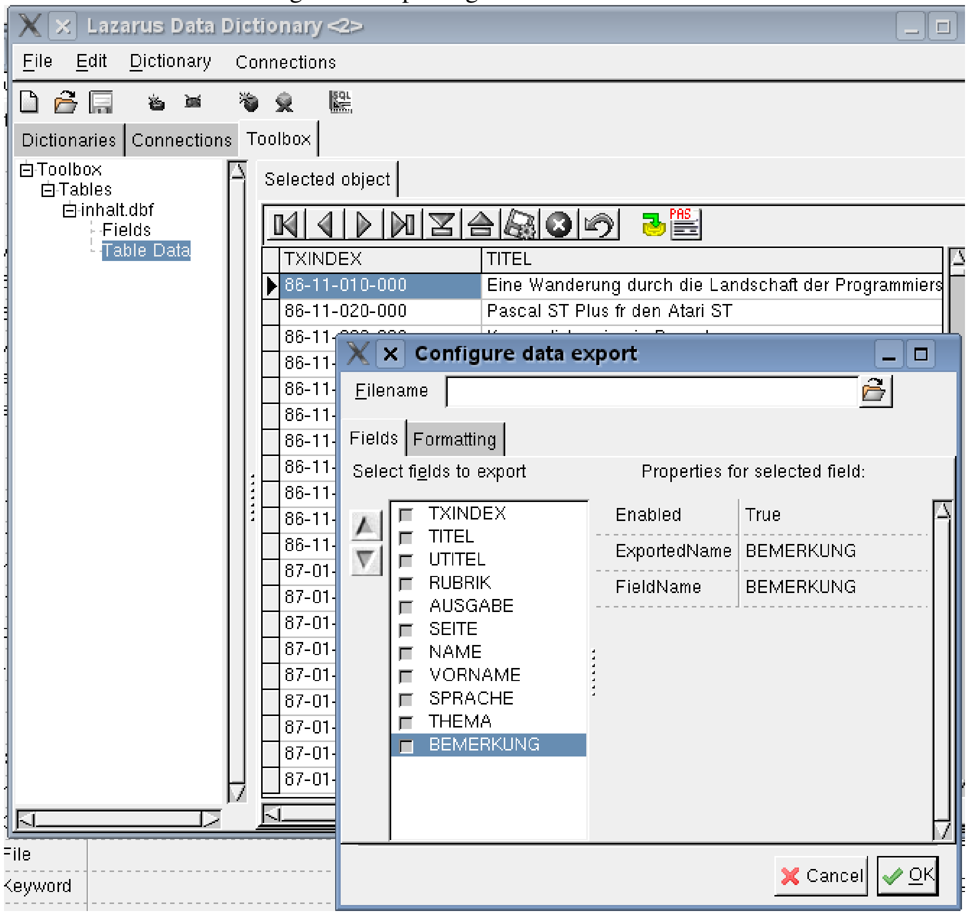
The export mechanism can be applied to the Toolbox data: choosing the XML export format will result in a export configuration dialog as in figure 6 on page 9. To the left is the list of fields that can will be exported. Each field can be disabled, so it won't be included in the export. To the right, a small grid with export opions for this field is shown: in the case of XML the exportedname is the name of the XML tag used to represent this field. The second tab shows some general export options: mostly controlling the format of the output. Obviously, the options are different for each of the export formats. Finally, the filename can be filled in on the top of the dialog: the data will be exported to the filename given there.

The export mechanism can also be used in Lazarus applications: installing the lazDBExport export will put a lot of components on the component palette, under 'Data Export': an export component for each of the data formats, plus a component (`TFPDataExporter`) which will show a selection and configuration dialog for each of the export formats compiled into the application.

To register the standard export formats in the application, a `TStandardExportFormats` component exists: it can be dropped on a form, and the desired formats can be made available to the end-user by setting a simple property. Other custom-made formats need to be registered in code.

The Toolbox application can be used to demonstrate this: dropping the `TStandardExportFormats` component and the `TFPDataExporter` component (name it `DEToolbox`) is all that is required: the `Dataset` property of the `DEToolbox` component should obviously be set to the DBF file. Lastly, a button `BExport` is added with the following `OnClick` handler:

Figure 6: Exporting the toolbox contents



Toolbox
Tables
inhalt.dbf
Fields
Table Data

TXINDEX	TITEL
86-11-010-000	Eine Wanderung durch die Landschaft der Programmiers
86-11-020-000	Pascal ST Plus fr den Atari ST
86-11-	
86-11-	
86-11-	
86-11-	
86-11-	
86-11-	
86-11-	
86-11-	
86-11-	
87-01-	
87-01-	
87-01-	
87-01-	
87-01-	
87-01-	
87-01-	
87-01-	
87-01-	

File
Keyword

Filename

Select fields to export

- TXINDEX
- TITEL
- UTITEL
- RUBRIK
- AUSGABE
- SEITE
- NAME
- VORNAME
- SPRACHE
- THEMA
- BEMERKUNG

Properties for selected field:

Enabled	True
ExportedName	BEMERKUNG
FieldName	BEMERKUNG

```
procedure TToolBoxForm.BExportClick(Sender: TObject);
begin
  DEToolbox.Execute;
end;
```

That all that is needed to add export functionality to any DB-Aware application.

8 Creating code

The second functionality offered by the data desktop is to create Object Pascal code for various purposes. This can range from very simple to complicated code. Currently, the following code can be created:

1. Transform an SQL statement to a Object Pascal string constant.
2. Based on a dataset (a query result or table data), create code to create a DBF file that can be used to keep the shown data.
3. Based on a dataset, create a class with properties corresponding to the fields in the data. Additionally, code can be generated to load the properties from a dataset.
4. Similar to the previous item, based on a dataset, create a TiOPF class declaration with properties corresponding to the fields in the data, and create the necessary visitor classes to persist the class to a database.

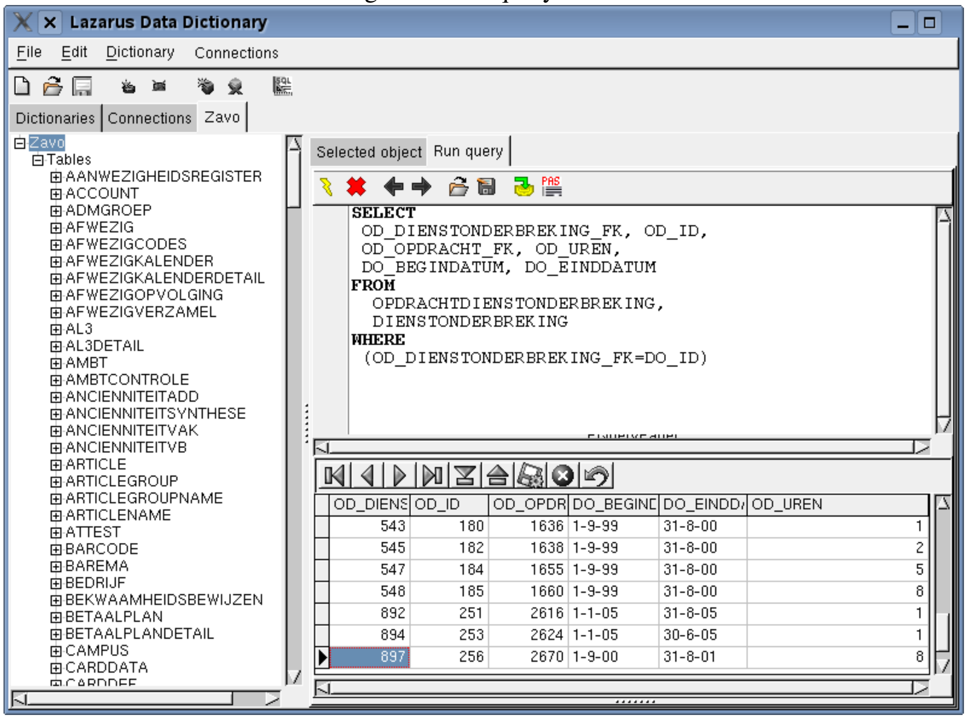
These code generators are delivered standard with Free Pascal. As with the export formats, custom code generators can be written and registered in the system. When selecting the 'Generate code' functionality, a choice of available code generators is presented, and when a generator is chosen, a configuration dialog similar to the export configuration dialog is presented to the user.

As an example, the toolbox table can be represented by the following class:

```
TMyObject = Class(TPersistent)
Public
  Procedure Assign(ASource : TPersistent); override;
Published
  Property TXIndex : AnsiString Read FTXIndex Write FTXIndex;
  Property Titel : AnsiString Read FTitel Write FTitel;
  Property UTitel : AnsiString Read FUTitel Write FUTitel;
  Property Rubrik : AnsiString Read FRubrik Write FRubrik;
  Property Ausgabe : AnsiString Read FAusgabe Write FAusgabe;
  Property Sprache : AnsiString Read FSprache Write FSprache;
  Property Name : AnsiString Read FName Write FName;
  Property Vorname : AnsiString Read FVorname Write FVorname;
  Property Thema : AnsiString Read FThema Write FThema;
  Property Bemerkung : AnsiString Read FBemerkung Write FBemerkung;
end;
```

The private part of the class was skipped. The complete generated code can be found in the otoolbox.pp file on the CD accompagnying this issue. By default, the generated code should be compilable without any editing. It is of course possible to specify a combination of options that does not compile without additional editing; some basic checking for the validity of the given options is done, but not all cases can be covered: for instance, it is possible to specify a non-existing custom type for the type of a property.

Figure 7: The query editor



```
SELECT
 OD_DIENSTONDERBREKING_FK, OD_ID,
 OD_OPDRACHT_FK, OD_UREN,
 DO_BEGINDATUM, DO_EINDDATUM
FROM
  OPDRACHTDIENSTONDERBREKING,
  DIENSTONDERBREKING
WHERE
 (OD_DIENSTONDERBREKING_FK=DO_ID)
```

OD_DIENS	OD_ID	OD_OPDR	DO_BEGINE	DO_EINDD/	OD_UREN
543	180	1636	1-9-99	31-8-00	1
545	182	1638	1-9-99	31-8-00	2
547	184	1655	1-9-99	31-8-00	5
548	185	1660	1-9-99	31-8-00	8
892	251	2616	1-1-05	31-8-05	1
894	253	2624	1-1-05	30-6-05	1
897	256	2670	1-9-00	31-8-01	8

Like the export functionality, all code generators can be used in lazarus programs as well: While not directly useful for end-user programs, it can ease the task of writing support tools.

9 Executing queries

On databases that are SQL based, the connection tab displays a query editor. This editor allows to create and execute queries. If the query is a select query, the result of the query will be shown: the same export and code generating mechanisms are available as when viewing table data. The query editor currently offers a history list, and saving/loading queries from disk. It should look like figure 7 on page 11.

An extra code generator available in the query editor is the generation of a SQL statement constant. If the following query is typed in the query editor:

```
SELECT
 OD_DIENSTONDERBREKING_FK, OD_ID,
 OD_OPDRACHT_FK, OD_UREN,
 DO_BEGINDATUM, DO_EINDDATUM
FROM
  OPDRACHTDIENSTONDERBREKING,
  DIENSTONDERBREKING
WHERE
 (OD_DIENSTONDERBREKING_FK=DO_ID)
```

Then the 'SQL constant' code generator can create the following Object Pascal constant:

```
Const
```

```
SQL = 'SELECT'+sLineBreak
      +' OD_DIENSTONDERBREKING_FK, OD_ID,'+sLineBreak
      +' OD_OPDRACHT_FK, OD_UREN,'+sLineBreak
      +' DO_BEGINDATUM, DO_EINDDATUM'+sLineBreak
      +'FROM'+sLineBreak
      +'  OPDRACHTDIENSTONDERBREKING,'+sLineBreak
      +'  DIENSTONDERBREKING'+sLineBreak
      +'WHERE'+sLineBreak
      +' (OD_DIENSTONDERBREKING_FK=DO_ID)';
```

Which can be directly copied and pasted in the Lazarus IDE.

10 Conclusion

The lazarus data desktop is a tool designed to facilitate the use of databases when working in Lazarus. While not a shrink-wrapped solution, it's extensible API allows to customize it to suit the needs of the programmer using it: each programmer has his own coding style and habits: one of the design goals of the database desktop is that any programmer should be able to add to it's functionality. At the same time, it aims to offer a default set of functionality likely to cover most common situations. This aim has not yet been achieved, many extensions are still planned:

- Support for various codepages: the observant user will have noted that the toolbox data is displayed not correct: this is because the data was saved in a DOS codepage, and is displayed on a display with UTF-8 codepage. This is currently under development.
- More database functionality: support for stored procedures, triggers, generators and external functions.
- More export formats: HTML, Excel, PDF etc.
- More code generators: e.g. creation of setup/teardown methods for fpcunit testcases.
- Tighter integration of the code generator in the Lazarus IDE.

Despite the fact that these aims have not yet been achieved, the database desktop already offers enough functionality to justify it's use when working with Databases and Lazarus.