Using AnyDac in Delphi

Michaël Van Canneyt

March 19, 2012

Abstract

Anydac is a data access layer for Delphi. It can be used to connect to many RDBMSes without the need for driver DLLs, all code is compiled-in to the application. Besides providing TDataset-based access to data, it also offers lots of other tools. A closer look.

1 Introduction

Most - if not all - programs need to access data in a database. Delphi programmers are spoiled with a wealth of data access technologies. Delphi itself ships with 2 general-purpose technologies: the (now deprecated) BDE and the recommended technology: dbExpress, which can be used to access data in SQL-based databases. Furthermore, Delphi provides access to ADO technology from Microsoft, providing access to even more databases.

If none of these technologies are satisfying, there is ZeosLib, an open source technology which provides access to many RDBMS engines as well.

In this article, an alternative to all these possibilities is presented: AnyDAC. AnyDAC is a data access technology, created by DA-Soft. It can be purchased from their website:

`http://www.da-soft.com/`

Anydac is a very rich set of components that can replace BDE, dbExpress, Zeos and ADO. It offers direct access to many database engines. Not only open source engines such as Firebird, Interbase, SQLIte, MySQL, PostgreSQL and Berkely DB, but also commercial engines: Oracle, DB2, MS-SQL server, MS-Access, Sybase and Blackfish SQL server. AnyDac also includes components for flat file databases and in-memory data, as well as moving data between any 2 supported engines.

Besides providing access to data in these RDBMSes, it also provides services for creating and restoring backups for engines that support this. For engines that support event notification, AnyDAC presents components to receive these events as well. For people that do not want or need to use TDataset descendents, AnyDAC provides low-level access to the engines it supports.

AnyDAC not only works for Delphi, it can be used in Free Pascal or Lazarus as well: Obviously only databases which have Linux client libraries are supported on Linux.

In this article, we will show how to use AnyDac to connect to a Firebird Database, but the techniques discussed are the same for any other database kind.

2 Installation

Installation in Delphi is simple and straightforward. An installer is provided which installs AnyDAC on Windows. The installer will present a list of supported drivers, and will offer to populate a sample database (the Northwind database). It is recommended to install at least one such database, as all examples require access to the sample database. It does not matter which RDBMS is used for this, but the client library for the selected RDBMs(es) must be installed, and a database must have been created before the installation procedure starts.

The installation procedure will offer to use synedit support in the GUI elements that show an SQL query. If you wish to make use of this, Synedit must be installed on the computer prior to installing AnyDac. If it is not installed or the installed synedit package name is not the correct one, installation of AnyDAC in the IDE will fail.

After AnyDAC is installed, the IDE will have 5 extra tabs:

AnyDAC These are the core components to access the databases and execute queries: `TADConnection`, `TADManager` and `TADQuery` are the most important components on this tab.

AnyDAC UI Contains some components that should be dropped on a form to enable some GUI features: a wait cursor, a login dialog and even a complete visual query builder dialog can be found on this tab.

AnyDAC links This tab contains a link component for every supported database: Dropping a link component on a form will include the driver for the database in the executable. It allows to set some options that are driver specific, such as the name of the client library to use. For example, dropping `TADPhysIBDriverLink` on the main form will include the firebird support in the program.

AnyDAC Services This tab contains components that provide database-specific services, such as backup and restore of a database. It also contains the event-notification components.

AnyDAC Devs This tab contains some extra components, which can be of general interest for programmers that use AnyDAC. They are not needed for general data access.

The installation comes with an 800 page manual, both in Windows-Help (.chm) as PDF format. It is worth reading at least the introductory chapters before starting to program in AnyDAC.

Besides installing the Delphi component on the Delphi component palette, several applications are also installed in the `Bin` directory of the installation:

ADAdministrator An administrator program for AnyDAC database definitions.

ADExplorer An SQL executing program for databases known in AnyDAC. It demonstrates lots of AnyDAC features.

ADMonitor A SQL Statement tracing program. It shows the SQL commands executed by the AnyDAC components.

Last but not least, there are a lot of sample programs: each aspect of AnyDAC is demonstrated in one or another example program. They are listed in the `Demos` project group in the Samples directory.

3 Architecture

AnyDAC consists of several layers, that are stacked on top of each other.

Physical layer The bottom layer is the physical layer: It defines an abstract API for physical data access, and contains actual implementations of this API: basically, the code which interacts with the RDBMS client library. It executes commands and transfers data from the Delphi application to the client library and vice versa.

Adaptor layer This layer sits on top of the physical layer and presents a unified API for database access. It can be used by itself, and contains lots of logic for creation and optimization of the SQL commands that are used by AnyDAC.

Component layer This layer contains the non-visual components that are actually used in the application. For example, the connection component `TADConnection` and query component `TADQuery` are in this layer.

GUI layer these are some GUI helper components such as the wait cursor and login dialog. They are plugins into the lower-level layers: The low-level API does not know how to show a wait cursor, but defines an interface for showing a cursor. Likewise, it does not know how to show a login dialog, but defines an interface for showing such a dialog. The GUI layer contains implementations for these interfaces.

There are some other smaller parts in AnyDAC, but they will rarely be used. In practice, one element of the physical layer will be used, and many components from the component layer.

4 Connecting to a database

After all the theory, it is time to show how AnyDAC works in practice by showing how to connect to a database.

For this, a small application will be created that connects to a tracker database: this database contains (fictitious) data about a school, which keeps track of when the pupils enter and leave the school premises. It has only 2 tables:

PUPIL This table contains the names of the pupils in the school (in fields `PU_FIRSTNAME` and `PU_LASTNAME`) and a primary key (`PU_ID`) (normally an autoincremental field).

PUPILTRACK This table contains a record for each time the pupil leaves or enters the school. The record contains a type field `PT_TYPE` containing 'I' or 'O' (for 'in' or 'out'). It has a timestamp, and of course a pointer to the record of the pupil.

The database will be created in Firebird, but any RDBMS can be used.

Much like the BDE (aliases) and ODBC (DSN), AnyDAC works with a central connection definition file. The central connection definition file is a file with definitions for all database connections about on the computer that the AnyDAC layer must know about.

It is an .ini file, containing a section for each known database connection. The section contains contains the necessary values to select the driver for the database, and to point the driver to the location of the database.

The following is a sample of such a section, defining a connection named 'Tracker':

```
[Tracker]
DriverID=IB
User_Name=MyUser
Password=MyPassword
Server=192.168.0.98
Database=/home/firebird/tracker.fdb
```

Visibly, the format is very simple. It should be easy for a program installer to create entries in this file. However, more options can be stored in this file than shown above. There are several ways to manage the contents of this file:

1. The `ADAdministrator` program that comes with the AnyDAC installation allows to manage the database definitions in the file. It is meant for the developer machine.
2. The `TADConnection` component editor can create an entry in this file. This is available only in the Delphi IDE.
3. The `TADManager` component can create entries in this file at runtime. It reads and writes entries to the file, and all options that can be stored in the file, can be specified as properties. This is by far the most powerful option.

To connect to the database in the IDE, a `TADConnection` component must be used. Let us call it `CTracker`. Double clicking the component will show the connection definition dialog. It is shown in figure 1 on page 5. After setting the necessary parameters the 'test' button can be used to test the connection. If the connection can be established, then pressing 'Save' will configure the connection component and the dialog can be closed.

Setting the `Connected` property of the component will activate it. Normally this explicit step is not needed: opening a dataset that is connected to this connection component, will open the connection.

If the connection was alredy defined in the connection file, then setting the `CTracker` components' `ConnectionDefName` property to the name of the definition should be sufficient to be able to connect to the database.

At runtime, on the computer where the application must be installed, the connection definition file may not be available, or the needed connection may not be defined. The `TADManager` component can be used to create the needed definition in code before an attempt is made to connect to the database.

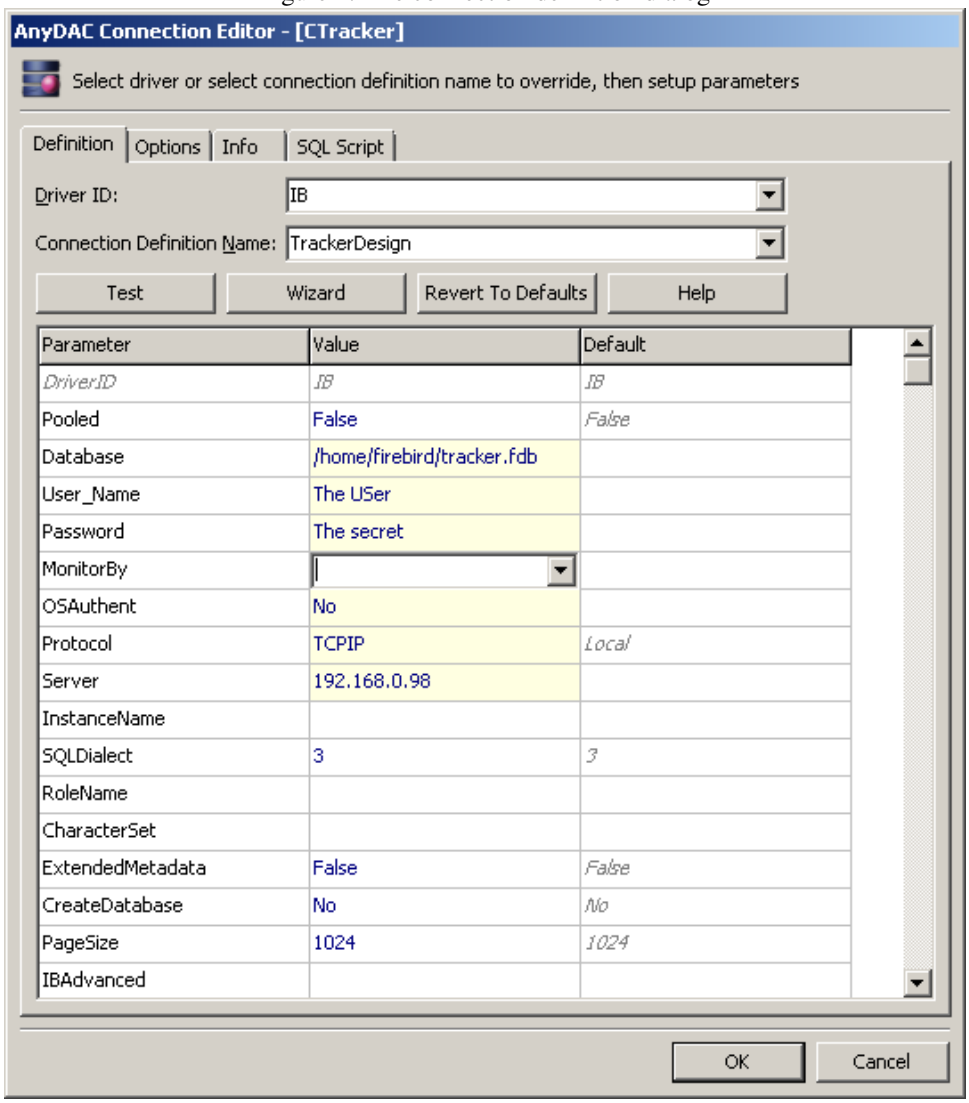
To cater for this, when the tracker application needs to connect to the database, the following code is executed:

```
procedure TForm1.CreateConnection;

Var
 I : IADStanConnectionDef;

begin
  I:=MGrAD.ConnectionDefs.FindConnectionDef('Tracker');
  if (I=Nil) then
    begin
    // create new connection
    I:=MGrAD.ConnectionDefs.AddConnectionDef;
    // Set the needed parameters
    I.DriverID:='IB';
    I.Name:='Tracker';
```

Figure 1: The connection definition dialog



```
    I.UserName:='myuser';
    I.Password:='mypassword';
    I.Server:='192.168.0.98';
    I.Database:='/home/firebird/tracker.fdb';
    // Optional.
    I.MarkPersistent;
    I.Apply;
    end;
end;
```

`MgrAD` is a `TADManager` component instance. The first line checks whether a `Tracker` connection definition can be found. If none is found, one is created. The properties speak for themselves: they need no explanation.

The last 2 lines are optional: the connection definition can be temporary. In that case it is not saved to the connection file, but disappears as soon as the program is ended. To save the connection definition to disk, the 'MarkPersistent' method will tell the `TADManager` component that the entry should be saved to disk. The call to `Apply` will actually write the definition.

After the connection is defined, the connection can be established:

```
procedure TForm1.Connect;

begin
  CTracker.ConnectionDefName:='Tracker';
  CTracker.Connected:=True;
end;
```

The first line points the connection component to the newly created connection definition. The second line actually opens the connection.

5 Fetching data from the database

Now that the application is connected to the database, data can be retrieved. The application will show a list of pupils in a grid. When a pupil is selected, it will show the tracking data for the selected pupil in a second grid below the first.

To fetch data from a database, AnyDAC offers 3 components.

TADQuery A TDataset descendant in which a select statement can be entered using an SQL statement.

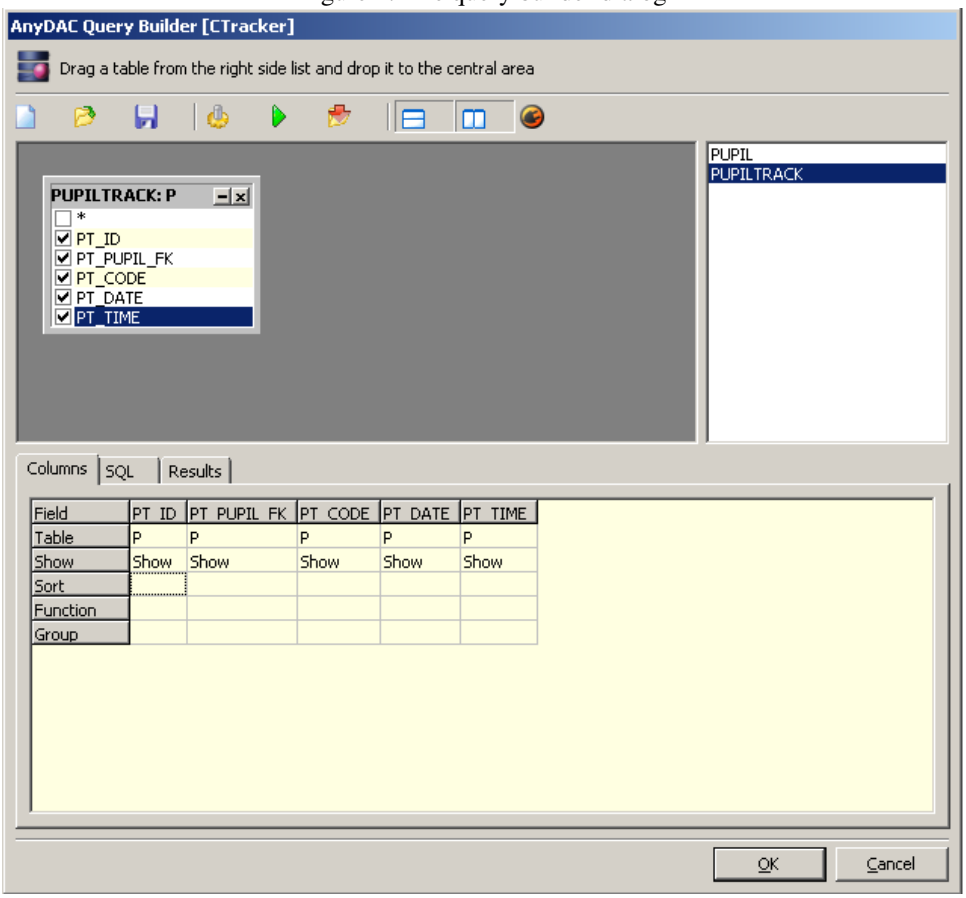
TADTable A TDataset descendant which a table name can be set, and it will fetch all data from the table.

TADStoredproc A TDataset descendant which a stored procedure name can be set, as well as parameters for the procedure. It will execute the stored procedure and fetch all data returned by the procedure.

These components are very similar to their equivalents in the DBX, BDE or ADO technologies.

From here on, the procedure to get data is pretty straightforward and not any different from the other data access technologies:

Figure 2: The query builder dialog



When dropping a `TADQuery` component on a form (we'll name it `QPupils`), its `Connection` property will automatically be set to the `CTracker` instance of `TADConnection`. The `SQL` property can be set to a simple

```
SELECT * FROM PUPIL;
```

Setting `Active` to `True` will open the dataset. Connecting a `TDatasource` instance (we'll name it `DSPupils`) to the dataset and linking a `TDBGrid` control to the datasource will display the pupils on the screen, in the IDE.

6 Using Parameters to create master/detail

Now that the pupils can be shown remains the task to show for each pupil the tracking data. This is a typical situation for a master/detail relation.

As one would expect from a Delphi data access technology, AnyDAC offers the use of parametrized queries and this can be used to create master/detail relations between datasets. Parameters can be specified in a `TADQuery` just as in the other data access technologies. To show this, we drop a second `TADQuery` instance which we name `QTrack`. Double clicking the component will bring up the SQL editor. The SQL editor supports visual query building, as shown in figure 2 on page 7. This editor can be used to create SQL statements in a visual way. We can use it to create the following statement:

```
SELECT
  P.PT_ID, P.PT_PUPIL_FK, P.PT_CODE, P.PT_DATE, P.PT_TIME
FROM
  PUPILTRACK P
WHERE
  (PT_PUPIL_FK=:PU_ID)
```

Note that the editor automatically creates a table prefix (P in this case) for all tables. The parameter in the SQL statement is specified in the usual way: `:PU_ID`. Its value can be set in code using the usual mechanism, for example:

```
  QTrack.Params.ParamByName('PU_ID').AsInteger:=1;
```

To create a true master-detail relationship, the `MasterSource` property of `QTrack` can be set to `DSPupils`. Now, when opening `QTrack`, it will fetch the value for the `PU_ID` parameter from the current record in `QPupils`. Additionally, when navigating in `QPupils`, the `QTrack` query will refetch the data as the current record in `QPupils` changes.

AnyDAC does not only support parameters, it also supports macros. Parameters can be used to execute the same SQL statement multiple times using different values for the parameter. AnyDAC uses the RDBMS' native support for parameters (if it supports that), making it very efficient. Macros, on the other hand, can be used to change the query itself. Macros are expanded before the query is sent to the RDBMS. In AnyDac, macros are indicated by the ! or & sign.

To demonstrate the use of macros, we implement a small feature in the tracker: when clicking on the header column in the grid showing pupils, the list of pupils should be ordered by the field shown in that column. Clicking a second time should revert the sort order.

To do this, the `SQL` property of `QPupils` can be set to:

```
SELECT * FROM PUPIL ORDER BY !SORTFIELD !SORTORDER
```

The query now has 2 macros: `SORTFIELD` and `SORTORDER`.

The `OnTitleClick` event of the grid showing pupils can be coded as follows:

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);

Var
  FN : String;
  MFN : TADMacro;
  MSO : TADMacro;

begin
  FN:=Column.FieldName;
  QPupils.Close;
  MFN:=QPupils.Macros.MacroByName('SORTFIELD');
  MSO:=QPupils.Macros.MacroByName('SORTORDER');
  if (FN=MFN.AsIdentifier) then
    begin
    // Same field, reverse sort order
    If MSO.AsRaw='ASC' then
      MSO.AsRaw:='DESC'
    else
```

```
      MSO.AsRaw:='ASC';
    end
  else
    begin
    // Other field. Set sort order to ASC
    MFN.Asidentifier:=FN;
    MSO.AsRaw:='ASC';
    end;
  QPupils.Open;
end;
```

The code is quite simple: The query is closed, and the macros are stored in local variables. After deciding whether the user clicked a new column or not, either the new value for the sortfield is set or the new sort direction is specified, and the query is opened again.

Note the use of `AsRaw` and `AsIdentifier`: The macros do not only support plain text substitution, but they also understand SQL, and each macro has a data type assigned to it: For `SortField` this is `mdIdentifier`, because it will contain an identifier. Setting `AsIdentifier` will then enclose the macro value in the identifier quote character if need be. Similarly, `mdRaw` means raw text substitution, and the value should be set as `AsRaw`.

If not all records are fetched from the server, then this is the easiest way of sorting the records in the grid. If all records are fetched, then it is faster to use the local indexes offered by `TADQuery`.

7 Editing data

Obviously, AnyDAC not only supports displaying but also updating of data in the dataset. In fact, the `UpdateOptions` property of `TADQuery` offers several options to control which operations (insert/update/delete) are allowed, and how these operations must be performed. `AnyDac` tries to do a good job of trying to create the necessary update SQL statements all by itself. However, to be able to update or delete records, it needs a unique key for the record: the names of the fields that make up this key must be specified somehow. The key fields can be established in 3 ways:

1. The default way is to fetch the unique key information from the database. This is what happens if `fiMeta` is included in the `fetchOptions.Items` property of the `TADQuery` component. By default, it takes the name of the first table in the FROM clause of the SQL select statement. The downside of this is that this requires extra queries when opening the dataset, and this is slow.
2. When using persistent fields, the `pfInKey` option in the `ProviderOptions` of the key fields can be set.
3. Specify the key field names in the `UpdateOptions.KeyFields` property (separated by semicolons). If this property is set, it takes precedence over the `pfInkey` flag of the fields.

The `fiMeta` option must be removed from the `fetchOptions.Items` property to be able to use the last 2 options - it is included by default.

The `UpdateOptions` property offers many options to control the update operations. If the default mechanisms of AnyDAC are not sufficient, an `UpdateObject` can be specified: here, the SQL statements to use in update operations can be set explicitly. Double clicking this component shows a dialog that allows you to quickly generate these statements.

8 Import/Export

Besides simple data access, AnyDAC offers many utilities. One of this is the `TADDataMove` component. It can be used to move data between 2 datasets, or between a CSV file and a dataset. All that needs to be done is set source and destination for the data. This can be used to quickly load data into a database, or to duimp data from the database.

For instance, the list of pupils can be easily exported. To do this, we drop a `TADDataMove` component on the form, name if `DMPupils` and set its `Source` property to `QPupils`. The `DestinationKind` is set to `skText`. To let the user choose a filename for the dump, we add a TSaveFileDialog component (`SDDump`). The actual dump can then be coded as follows:

```
procedure TForm1.ADumpPupilsExecute(Sender: TObject);

Var
  FN : String;
begin
  if not SDDump.execute then
    exit
  else
    FN:=SDDump.FileName;
  FADGUIxSilentMode:=True;
  With DMPupils do
    begin
    TextFileName:=FN;
    Execute;
    end;
end;
```

The code is quite straightforward. Setting `TextFileName` and calling `Execute` is all that is needed to create the CSV file.

The data move component can do more. It can also read data from the CSV file and insert it in the dataset. All that must be done is invert the source and destination properties. The sample program contains the code for this.

If not all fields are needed, then the `Mappings` property can be used to specify which fields to dump. The component can also create a log file, which is useful when reading data from file, especially combined with the ability to ignore exceptions (for instance when primary key violations occur).

9 Conclusion

AnyDAC is a very rich set of components for data access. It can without doubt replace any existing data access technology of Delphi: this article has just scratched the surface of the available functionality. Services, batch operations, alerts, monitoring, scripts, conditional preprocessing of SQL statements, local indexes and local datasets are some of the possibilities that have not been treated. We'll treat some of these in a future article.