Blaise Pascal

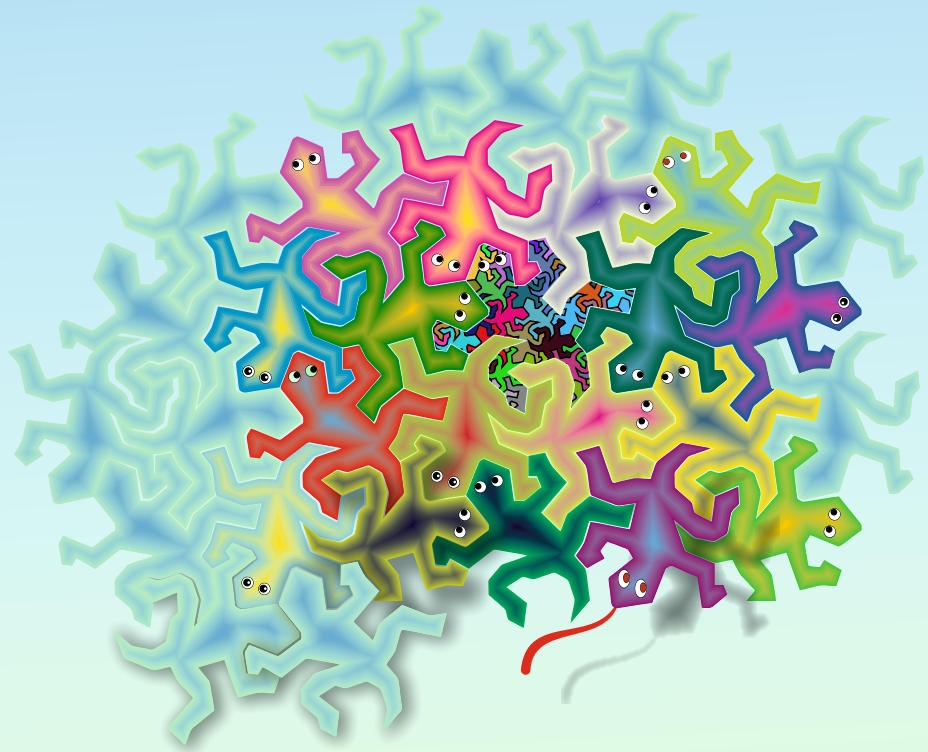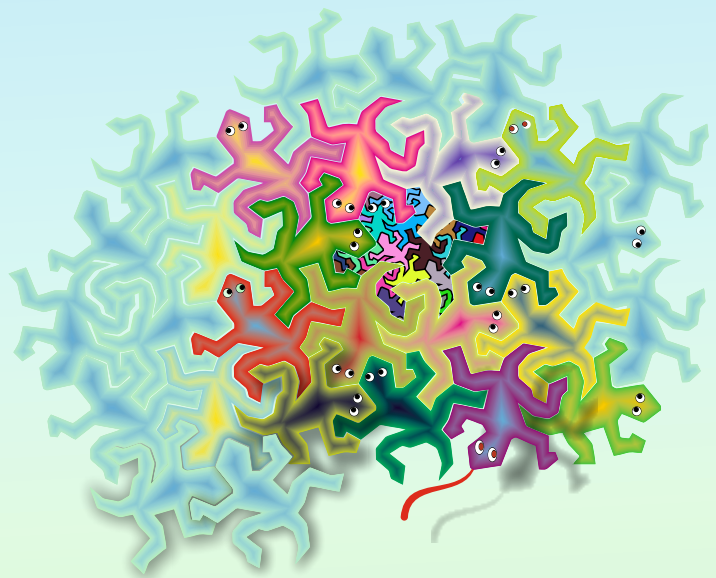# BLAISE PASCAL MAGAZINE 94/95

Multi platform /Object Pascal / Internet / JavaScript / WebAssembly / Pas2Js / Databases
CSS Styles / Progressive Web Apps
Android / IOS / Mac / Windows & Linux

Blaise Pascal

ARTICLES

ADVERTISERS

Pascal is an imperative and procedural programming language, which Niklaus Wirth designed (left below) in 1968–69 and published in 1970, as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. A derivative known as Object Pascal designed for object-oriented programming was developed in 1985. The language name was chosen to honour the Mathematician, Inventor of the first calculator: Blaise Pascal (see top right).

Niklaus Wirth

Contributors

Stephen Ball
http://delphiaball.co.uk
@DelphiABall

Peter Bijlsma -Editor
peter @ blaisepascal.eu

Dmitry Boyarintsev
dmitry.living @ gmail.com

Michaël Van Canneyt,
michael @ freepascal.org

Marco Cantù
www.marcocantu.com
marco.cantu @ gmail.com

David Dirkse
www.davdata.nl
E-mail: David @ davdata.nl

Benno Evers
b.evers @ everscustomtechnology.nl

Bruno Fierens
www.tmssoftware.com
bruno.fierens @ tmssoftware.com

Holger Flick
holger @ flixments.com

Primož Gabrijelčič
primoz @ gabrijelcic.org

Mattias Gärtner
nc-gaertnma@netcologne.de

Peter Johnson
http://delphidabbler.com
delphidabbler @ gmail.com

Max Kleiner
www.softwareschule.ch
max @ kleiner.com

John Kuiper
john_kuiper @ kpnmail.nl

Wagner R. Landgraf
wagner @ tmssoftware.com

Vsevolod Leonov
vsevolod.leonov@mail.ru

Andrea Magni www.andreamagni.eu
andrea.magni @ gmail.com
www.andreamagni.eu/wp

Paul Nauta PLM Solution Architect
CyberNautics
paul.nauta @ cybernautics.nl

Kim Madsen
www.component4developers.com

Boian Mitov
mitov @ mitov.com

Jeremy North
jeremy.north @ gmail.com

Detlef Overbeek - Editor in Chief
www.blaisepascal.eu
editor @ blaisepascal.eu

Howard Page Clark
hdpc @ talktalk.net

Heiko Rompel
info @ rompelsoft.de

Wim Van Ingen Schenau -Editor
wisone @ xs4all.nl

Peter van der Sman
sman @ prisman.nl

Rik Smit
rik @ blaisepascal.eu

Bob Swart
www.eBob42.com
Bob @ eBob42.com

B.J. Rao
contact @ intricad.com

Daniele Teti
www.danieleteti.it
d.teti @ bittime.it

Anton Vogelaar
ajv @ vogelaar-electronics.com

Danny Wind
dwind @ delphicompany.nl

Siegfried Zuhr
siegfried @ zuhr.nl

From your editor

Looks like we are finally getting the virus under control,
maybe we can get back to more normal circumstances.
I have been working like most of you very hard on our new projects:
end of the year we will publish Blaise Pascal Magazine 100,
which I think is a milestone.
So we are preparing to do something very special for all of our readers.
We of course will try to organize a party, and writing about that
we soon will organize (plan) a real meeting event in the Netherlands
so we can meet each other again.

This double issue has a lot of special subjects, very often about programming for the internet.
Like: a game which is a colouring chameleon page, created in Lazarus, you can find it on our website at:
`https://www.blaisepascalmagazine.eu/colorgame/`
and the **Latest Version of Free TMS Webcore for macOS/Linux/Windows** at page 65
and the article **New Pas2Js: Lazarus Webform , implementing API's for Chromium** at page 50,
and **Web Service Part 3** at page 97.

Mentioning the Internet brings up the progress **Martin Friebe** has achieved by embedding the Website Form for Chromium and we soon will have the first trials available.

Still a lot of work.

- but we see a probability before the end of the year that you will be able to create with Lazarus - and design your own website in "**What you see is what you get"** mode:
and also added: an
Object Inspector including
CSS ability,
Code Completion for
    CSS and JavaScript and a
Debugger for that site, coding in the browser etc.

Now I'm impatiently waiting to be able to implement **WebAssembly** in **Pas2Js**.
We planned to create a book and lessons how to use **Pas2JS.**
I will of course announce that.

Mattias Gärtner is very hard working on the new version for Lazarus and Michael van Canneyt and the team will try to update to the next version of FPC.

The latest news is that I am writing a book about **FastReport** for the use and completed with lessons and examples.

Now as ever:
if you have any ideas, comments or suggestions or ideas, let me know…

"We chameleons, like computers, have a ton of components. Luckily, we don't crash, get hacked or need constantly updated."

maXbox starter 85

There are two kinds of data scientists:
1) Those who can extrapolate from incomplete data.
2.....

Reading json data in maXbox or Lazarus should be easy with the right class. Json data can be read from a string, file or it could be a Json web link see later on.

But what's Json JSON (**JavaScript Object Notation**) is a lightweight data-interchange format. It is easy for humans to read and write at least as a text. It is easy for machines to parse and generate, but not so easy to interpret for humans.

Let's start with a simple sample:

```pascal
stjson:= '{"data":{"results":[{"Branch":"ACCT590003"}]}}';
```

First we create an object and parse it:

```pascal
ajt:= TJson.create();
ajt.Parse(stjson);
```

Now in Json4Delphi we can ask the type:

```pascal
writeln(botostr( ajt.IsJsonObject( stjson)));
writeln(botostr( ajt.IsJsonString( stjson)));
writeln(botostr( ajt.IsJsonArray( stjson)));
cnode:= ajt.JsonObject.items[0].name;
writeln(cnode)
```

```
        TRUE
        FALSE
        FALSE
        data
```

As you can see the sample is an object node and data is the cnode.
JSON for Delphi supports also older versions of Delphi (7 or above) and its Object Pascal native code, using classes like TList, TStrings and TStringList is a great adavantage for speed, scripting and comprehension.
So how do we get the branch in our example:

```pascal
writeln('branch of data:
'+ajt['data'].asobject['results'].asarray[0].asObject['Branch'].asstring);
        branch of data: ACCT590003
```

So the branch is an object-array. Arrays in JSON are almost the same as arrays in Pascal or C. In JSON, array values must be of type string, number, object, array, boolean or null. In JavaScript, array values can be all of the above, plus any other valid JavaScript expression, including functions, dates or undefined. In Delphi we use of course strong types with overloading functions not dynamic string types!

```pascal
type
TJsonValueType = (jvNone, jvNull, jvString, jvNumber,
     jvBoolean, jvObject, jvArray);
TJsonStructType = (jsNone, jsArray, jsObject);
TJsonNull = (null);
TJsonEmpty = (empty);
```

On the other side JSON is a text format for representing objects and arrays, there is no such thing as a "JSON object" like a Object Pascal Object. Therefore we have to find out in our J4D library the type from the formal syntax:

```pascal
function TJsonBaseAnalyzeJsonValueType(
                const S: String): TJsonValueType;
var Len: Integer; Number: Extended;

begin
 Result:= jvNone;
 Len:= Length(S);
 if Len >= 2 then begin
  if (S[1] = '{') and (S[Len] = '}')
        then Result := jvObject
  else if (S[1] = '[') and (S[Len] = ']')
        then Result := jvArray
  else if (S[1] = '"') and (S[Len] = '"')
        then Result := jvString
  else if SameText(S, 'null') then Result := jvNull
  else if SameText(S, 'true') or SameText(S, 'false')
        then Result:= jvBoolean
  else if FixedTryStrToFloat(S, Number)
        then Result := jvNumber;
 end
 else if FixedTryStrToFloat(S, Number)
        then Result := jvNumber;
end;
```

Next topic is a Json-tree. Normally the packed collection data we use is imported from a file or folder but we can also parse and stringify a const as json4delphi data or test data:

```pascal
Const StrJson=
'{ '+
'   "destination_addresses" : [ "Paris, France" ], '+
'   "origin_addresses" : [ "Amsterdam, Nederland" ], '+
'   "rows" : [ '+
'       {    '+
'         "elements" : [  '+
'             {  '+
'               "distance" : { '+
'                   "text" : "504 km", '+
'                   "value" : 504203    '+
'               },  '+
'               "duration" : {  '+
'                   "text" : "4 uur 54 min.",  '+
'                   "value" : 17638  '+
'               },  '+
'               "status" : "OK"  '+
'           }  '+
'         ]  '+
'       } '+
'   ],  '+
'   "status" : "OK"  '+
'}';
```

Again we can see the formal syntax. Similar to other formed programming languages, an Array in JSON is a list of items surrounded in square brackets ([]). Each item in the array is separated by a comma.

A JSON object (a string to parse you remember) is a key-value data format that is typically rendered in curly braces{}. Our JSON object above looks something like this:

```
        {    '+
'              "distance" : { '+
'                 "text" : "504 km", '+
'                 "value" : 504203    '+
'              },    '+
'              "duration" : {    '+
'                 "text" : "4 uur 54 min.",
'+
'                 "value" : 17638    '+
'              },   '+
'       "status":"OK" '+
'          }
```

JSON arrays are ordered collections and can contain values of different data types and this is more flexible than in XML. I don't think that JSON syntax is very complicated and I prefer it over XML and YAML.

Ok. let's do two ways of accessing our distance map data from above:

```
  ajt:= TJson.create();
  ajt.Parse(StrJson);

  writeln(botostr( ajt.IsJsonObject(StrJson)));
  writeln(botostr( ajt.IsJsonString(StrJson)));
  writeln(botostr( ajt.IsJsonArray(StrJson)));
  writeln('get third name: '+ jt.JsonObject.items[2].name);
  writeln('get four name: '+ ajt.JsonObject.items[3].name);
  println('dist: '
+ajt['rows'].asarray[0].asObject['elements'].asarray[0].
asobject['distance'].asobject['text'].asstring);

get third name: rows
get four name: status
dist: 504 km
```

We can also access array or multi-dimensional array values by using a for loop and index numbers:

```
jOb:= ajt.JsonObject; //reference passing
  for cnt:= 2 to jOb.count-2 do begin
    Clabel:= job.items[cnt].name;
    writeln('iterate: '+clabel)
    JsArr:= job.values[Clabel].asArray;
    for cnt2:= 0 to jsarr.count-1 do
      jsobj:= jsarr.items[cnt2].asobject;
      for cnt3:= 0 to jsobj.count do
        writeln(jsobj['elements'].asarray[0].asobject.
        items[cnt3].name)
  end;
ajt.Free;

    iterate: rows
    distance
    duration
```
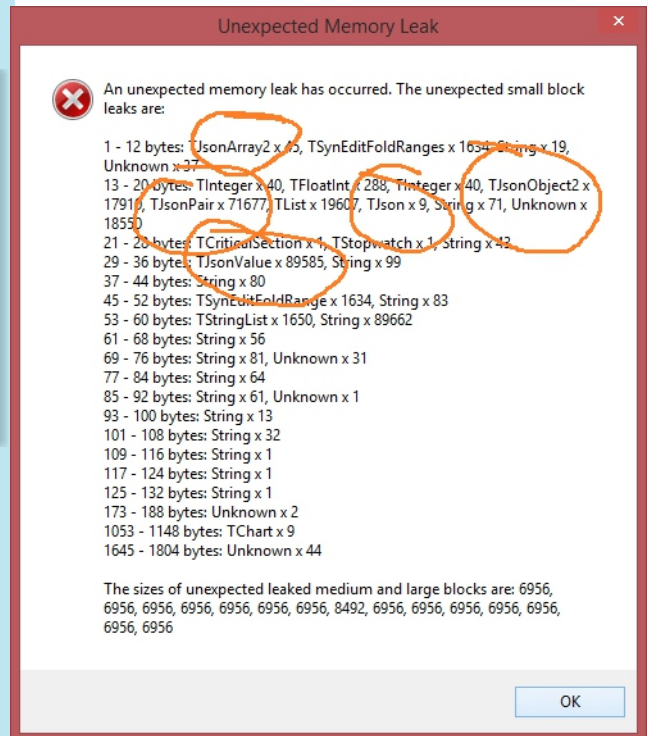
If you prefer direct access for example of the status:

```
println('elements status:
'+ajt['rows'].asarray[0].asObject['elements'].asarray[0
].asobject['status'].asstring);

      elements status: OK
```

For a big data collection it's important to know your memory allocation and free them as many as possible or keep the object lifetime short:

As a next and last sad example we get the data from web.

Let us first try to read the Json data from a web link.

```
Const
  JsonUrl = 'https://pomber.github.io/covid19/timeseries.json';
```

Now we need a Load URL() or Upload File() function to get the json data for parsing. In our case load is a ole automation function-pair of open and send(). We define the necessary packages "msxml2.xmlhttp" and the JSON class itself:

```
var XMLhttp:OleVariant; // As Object
           ajt: TJson; JObj: TJsonObject2;

XMLhttp:= CreateOleObject('msxml2.xmlhttp')
XMLhttp.Open ('GET', JsonUrl, False)
ajt:= TJson.create();
```

Let us import the covid19 timeseries data from this already mentioned Json link:
pomber.github.io/covid19/timeseries.json
using XMLhttp:

```
Ref: <class 'pandas.core.frame.DataFrame'>
RangeIndex: 82661 entries, 0 to 82660
Data columns (total 5 columns):
 #  Column    Non-Null Count Dtype
--- ------    -------------- -----
 0  country   82661 non-null object
 1  date      82661 non-null object
 2  confirmed 82661 non-null int64
 3  deaths    82661 non-null int64
 4  recovered 82661 non-null int64
dtypes: int64(3), object(2)
memory usage: 3.4+ MB
```

A Json Parser is then used to format the Json data into a properly and readable Json Format with curly brackets. That can easily view and identify its key and values. To get the Json type of class, struct or array, we need to use
`ajt.parse()` method first.
For slicing (filter) the data we copy the range from response timeseries.json:

```
start:= pos('"'+ACOUNTRY+'"',response);
stop:= pos('"'+ACOUNTRY2+'"',response);
writeln('Len Overall: '+itoa(length(response)))
resrange:= Copy(response, start, stop-start);
resrange:= '{'+resrange+'}';
writeln('debug sign on pos: '+GetWordOnPos(response, posex('],',response,1)));
try
  ajt.parse(resrange);
except
  writeln( 'Exception: <TJson>"" parse error: {'+
       exceptiontostring(exceptiontype,exceptionparam))
end;
Split(ajt.Stringify,'{',slist)
writeln('Statuscode: '+(statuscode)+': '+'listlen '+itoa(slist.count));
```

Now we can iterate through the keys with values as items. Here, in the above sample Json data: date, confirmed, deaths and recovered are known as key and "2020–1–22", 0, 0 and 0 known as a Value. All Data are available in a Key and value pair.
First we get a list of all 192 country names as the node name:

```
JObj:= ajt.JsonObject;
writeln('Get all Countries: ')
for cnt:= 0 to jobj.count-1 do
writeln(Jobj.items[cnt].name);
...United Kingdom
Uruguay
Uzbekistan
Vanuatu
Venezuela
Vietnam...
```

So the country is an object to get. Ok, it is a JsonObject dictionary with 192 countries. We check the keys of our dict with a nested loop of all confirmed cases:

```
for cnt:= 0 to Jobj.count-1 do begin
 Clabel:= Jobj.items[cnt].name;
 JArray2:= jobj.values[Clabel].asArray;
 for cnt2:= 0 to jarray2.count-1 do
  itmp:= jarray2.items[cnt2].asObject.values['confirmed'].asinteger;
end;
```

In a second attempt we visualize the timeseries with TeeChart Standard. Ok we got the object-array as sort of dataframe with items and values but not in the form that we wanted. We have to unwind the nested data like above to build a proper dataframe with series at runtime for Tchart:

```
//*) //accumulated  Clabel:='Vietnam';
Chart1.Title.Text.clear;
Chart1.Title.Text.add('Sciplot TimeSerie for: '+Clabel);
  JArray:= ajt.values[Clabel].asarray;
  writeln('jitems country '+itoa(jarray.count));
  for cnt:= 1 to jarray.count-1 do begin
      itmp:= jarray.items[cnt].asObject.values['confirmed'].asinteger;
      chart1.Series[0].Addxy(cnt,itmp,'',clGreen);
      itmp:= jarray.items[cnt].asObject.values['deaths'].asinteger;
      chart1.Series[1].Addxy(cnt,itmp,'',clRed);
      itmp:= jarray.items[cnt].asObject.values['recovered'].asinteger;
      chart1.Series[2].Addxy(cnt,itmp,'',clBlue);
  end;
```
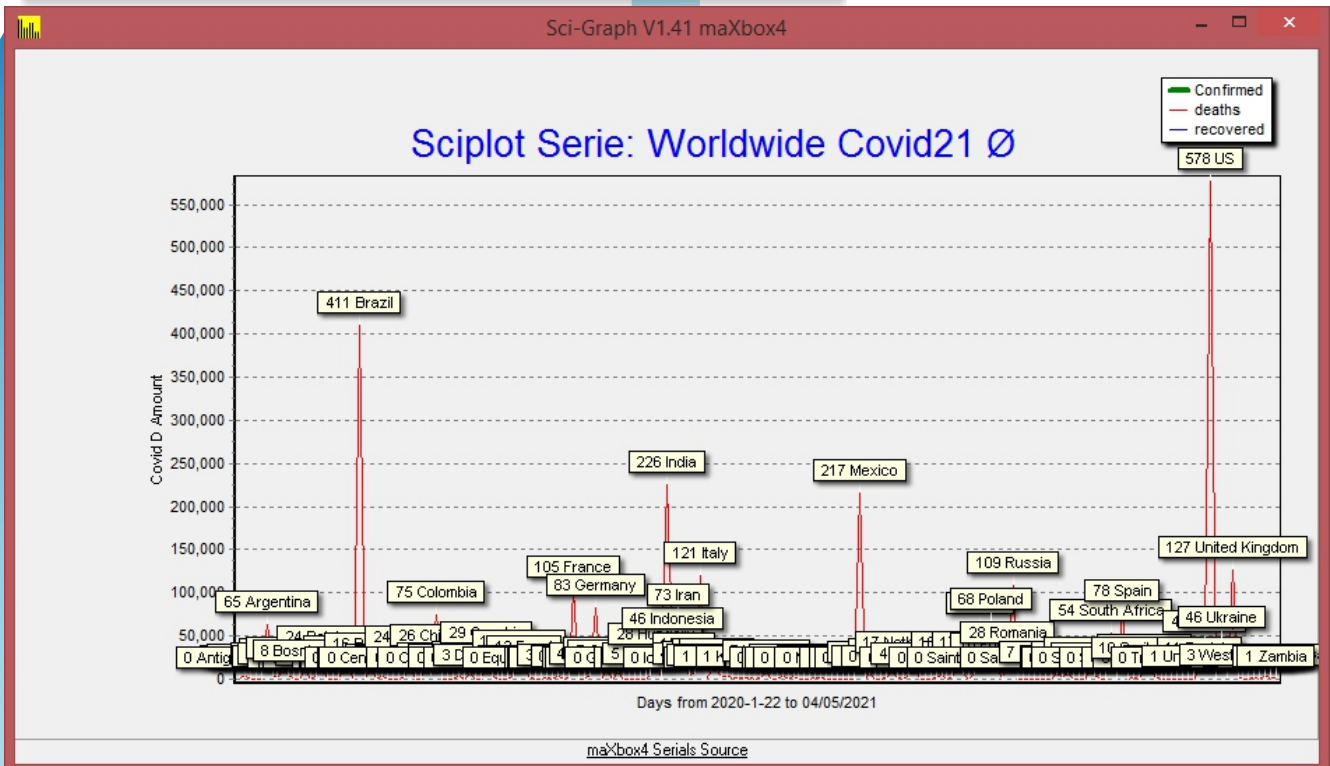

TeeChart is a charting library for programmers, developed and managed by Steema Software of Girona, Catalonia, Spain. It is available as commercial and non-commercial software. TeeChart has been included in most Delphi, Lazarus and C++Builder products since 1997, and TeeChart Standard currently is part of Embarcadero RAD Studio 10.4 Sydney.

Sci-Graph V1.41 maXbox4

Sciplot Serie: Worldwide Covid21 confirmed not †
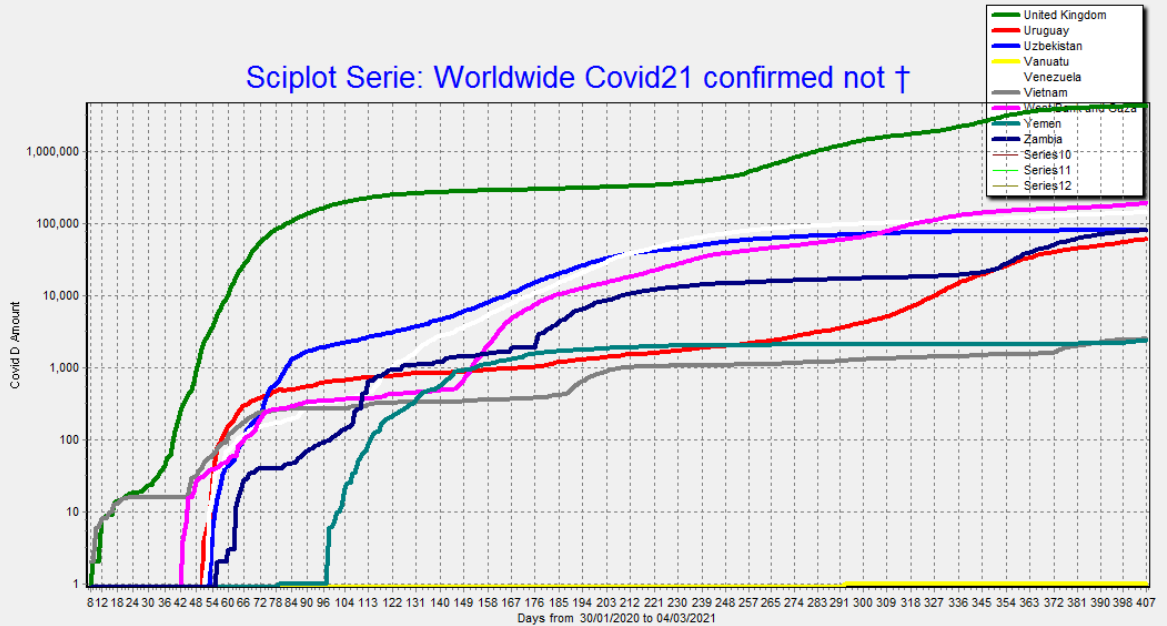
Choosing a Series Type for a Chart will very much depend on your own requirements for the Chart. There are occasions, however, where the choice of Chart depends on which Series types support the number of input variables because of the high number of variables to plot.

Sci-Graph V1.41 maXbox4

Sciplot Serie: Worldwide Covid21
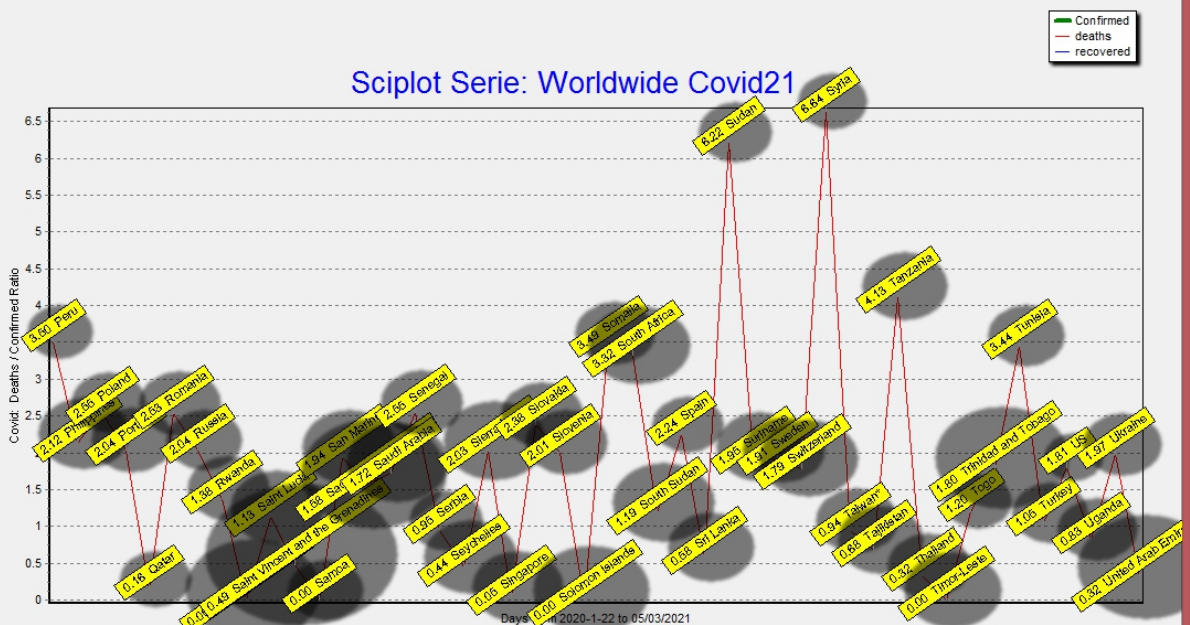
CONCLUSION:

The proper way to use JSON is to specify types that must be compatible at runtime in order for your code to work correctly. The `TJsonBase= class(TObject)` and `TJsonValue= class(TJsonBase)` namespace contains all the entry points and the main types. The `TJson= class(TJsonBase)` namespace contains attributes and APIs for advanced scenarios and customization.

JSON is a SUB-TYPE of text but not text alone. Json is a structured text representation of an object (or array of objects). We use JSON for Delphi framework (json4delphi), it supports older versions of Delphi and Lazarus (6 or above) and is very versatile. Another advantage is the Object-Pascal native code, using classes only `TList`, `TStrings`, `TStringStream`, `TCollection` and `TStringList`; The package contains 3 units: `Jsons.pas`, `JsonsUtilsEx.pas` and a project `Testunit`, available at: `https://github.com/rilyu/json4delphi`

The script can be found:

`http://www.softwareschule.ch/examples/covid2.txt`
`http://www.softwareschule.ch/examples/972_json_tester32.txt`

Ref:

`https://wiki.freepascal.org/TAChart_Demos`

`https://github.com/rilyu/json4delphi`

`https://github.com/rilyu/json4delphi/blob/master/test/TestJson.dpr`

`http://www.softwareschule.ch/examples/covidapp3.txt`

Doc: `https://maxbox4.wordpress.com`

Appendix: import register log from maXbox4 integration

```
{*------------------------------------------------------------------------*)
```

```
procedure SIRegister_Jsons(CL: TPSPascalCompiler);
begin
  CL.AddTypeS('TJsonValueType', ,(jvNone,jvNull,jvString,jvNumber,jvBoolean,jvObject,jvArray )');
  CL.AddTypeS('TJsonStructType', '( jsNone, jsArray, jsObject )');
  CL.AddTypeS('TJsonNull', '( jsnull2 )');
  CL.AddTypeS('TJsonEmpty', '( jsempty )');
  SIRegister_TJsonBase(CL);
  CL.AddClassN(CL.FindClass('TOBJECT'),'TJsonObject2');
  CL.AddClassN(CL.FindClass('TOBJECT'),'TJsonArray2');
  SIRegister_TJsonValue(CL);
  SIRegister_TJsonArray2(CL);
  SIRegister_TJsonPair(CL);
  SIRegister_TJsonObject2(CL);
  SIRegister_TJson(CL);
end;
```
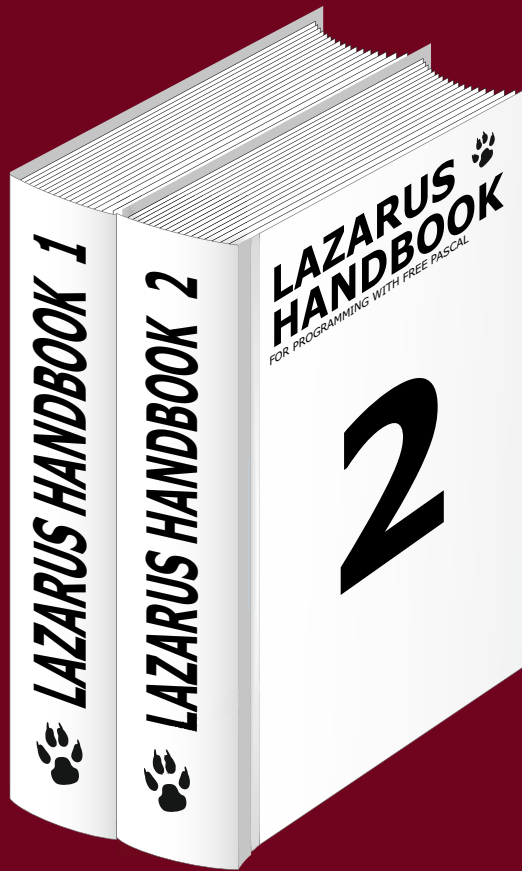
INTRODUCTION

When it comes to bringing web applications to a desktop environment the choice falls on **Electron** as it is the most popular framework among web developers.

Support for creating **Electron** cross platform desktop applications is included in **TMS WEB Core** for quite a while, but the downside is always there: it is a 3rd party solution that we have no control over.

B e i n g convinced that we could expose more desktop integration functionality including in particular local database access support, TMS software embarked on the development of an alternative framework under the name

MILETUS

The Miletus application template type in the Delphi IDE
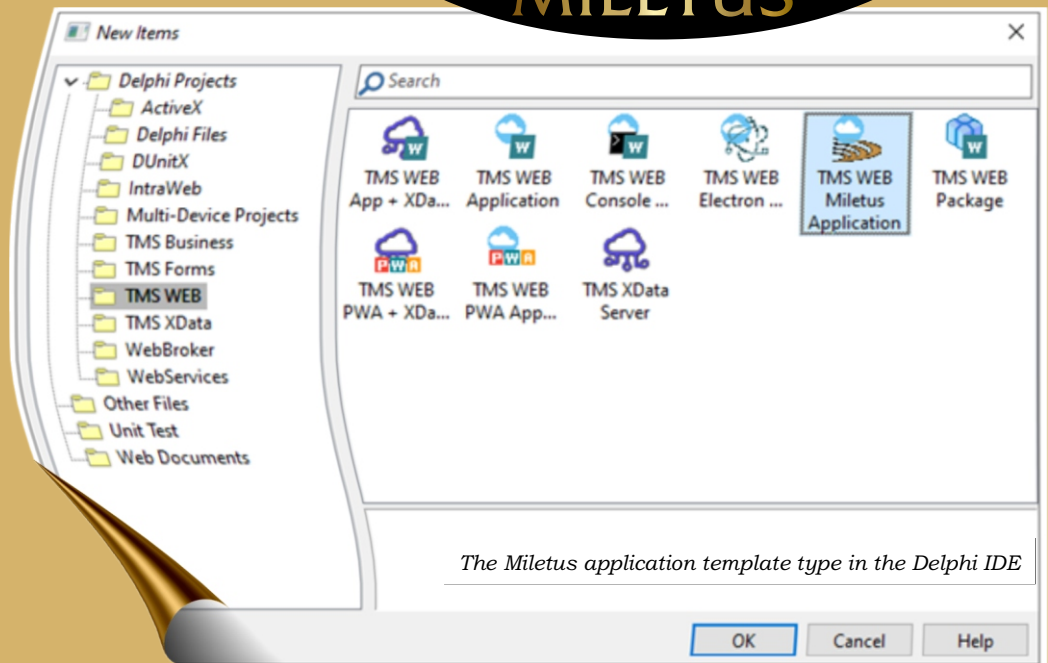
WHAT IS MILETUS EXACTLY?

Named after Thales of Miletus, Miletus is a framework that enables TMS WEB Core applications to run as desktop applications and it also provides access for native features. In a nutshell, you can fully reuse code created for a web application, take advantage of HTML/CSS for creating a modern & spectacular responsive user interface, access local files & local databases or closely integrate with operating system capabilities and still easily deploy the application as standalone executable. Recapitulating, pick the Miletus project type from the IDE, build the application and compile it and you get a resulting executable for Windows, macOS or Linux.

MILETUS

starter   expert

In the project manager in the Delphi IDE, it is shown as:
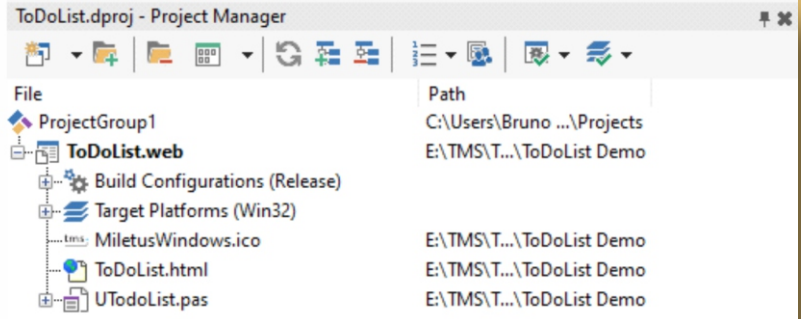
and on the form designer in the Delphi IDE we have:
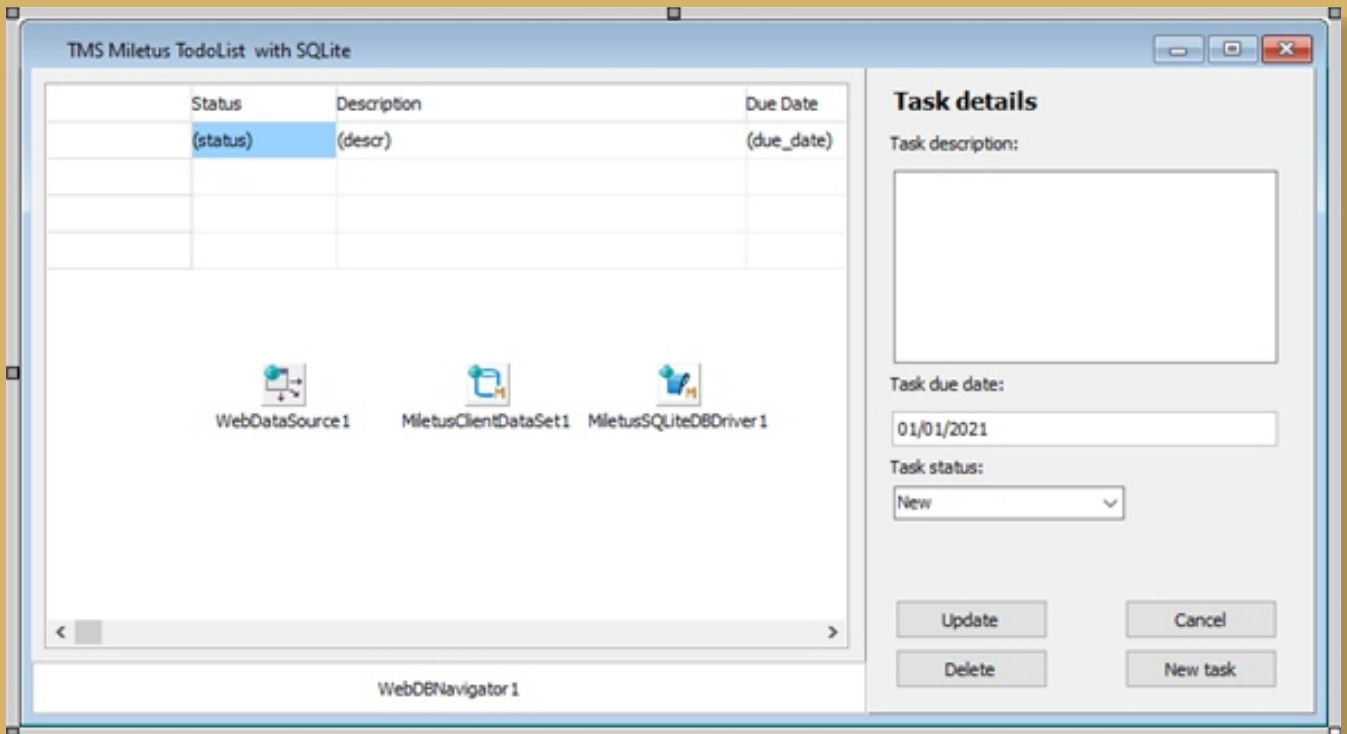
Thales of Miletus
before christ
(624/ 623 – 548/ 545 BC) was a Greek
mathematician, astronomer and pre-Socratic
philosopher from Miletus in
Ionia, Asia Minor. He was one of the Seven Sages of Greece. Many, most notably Aristotle,
regarded him as the first philosopher in the Greek tradition, and he is otherwise historically
recognized as the first individual in Western civilization known to have entertained and
engaged in scientific philosophy.
In mathematics, Thales used geometry to calculate the heights of pyramids and the distance
of ships from the shore. He is the first known individual to use deductive reasoning applied to
geometry, by deriving four corollaries to Thales' theorem. He is the first known individual to
whom a mathematical discovery has been attributed.

WIKIPEDIA

Compiling and running the application results in:
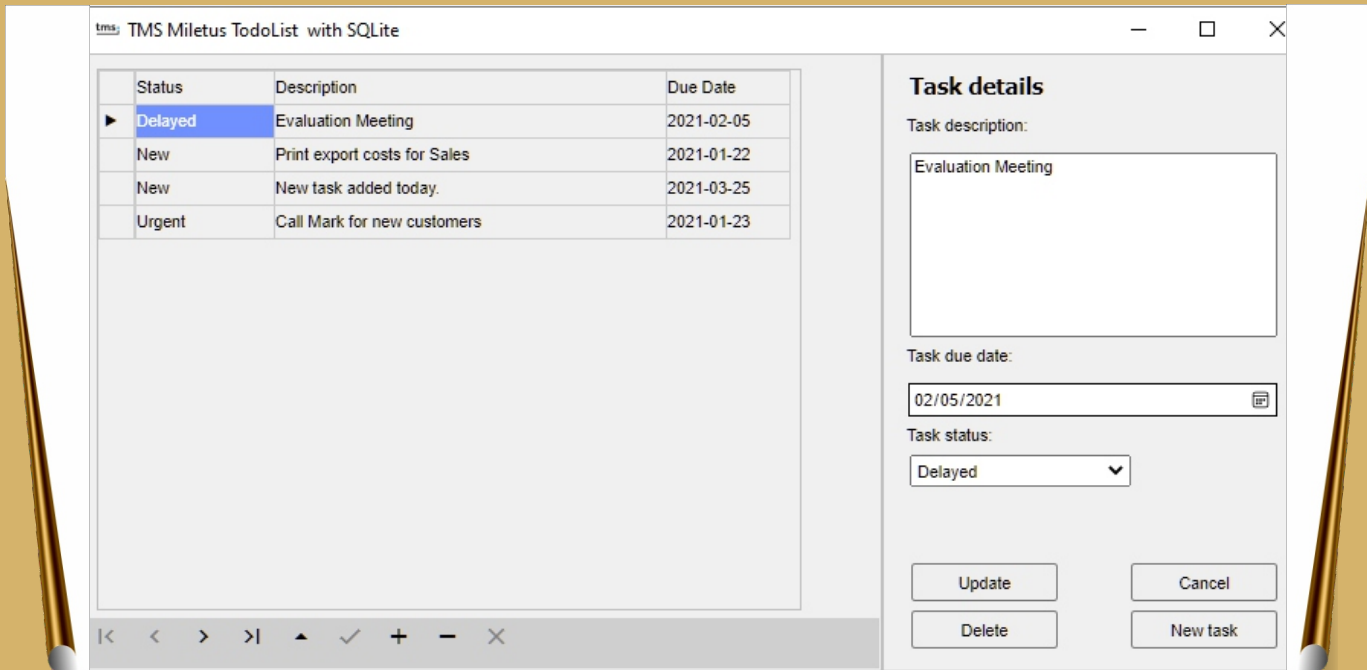
In the output folder, all we have is the generated single EXE application (Windows here) and the app can be directly deployed and started this way:

MILETUS ARCHITECTURE

A Miletus application consists of a web client application running in a browser that is hosted in a native shell application. There is a communication bridge between the web client application and the native shell application and this is what enables direct access to operating system functionality from the web application. A Miletus application wraps the default operating system browser in the native shell application, hence, this is Edge Chromium on Windows, Safari on Apple and Webkit on Linux.

MILETUS FRAMEWORK

Let's take a look at the available classes, functions and components that will enable you to interact with native operating system functionalities!

CLASSES AND FUNCTIONS

- **TMiletusStringList:**
  Read and write local text files.
- **TMiletusBinaryDataStream:**
  Similarly to **TMiletusStringList** this enables you to write and read local binary files. It also provides multiple formats to access the data.
- **TMiletusClipboard:**
  Read from and write to the OS clipboard.
- **TMiletusShell:**
  Exposes some shell functionalities: open a file with its default application, open an external URL with the default browser, move files to the trash and show files in the containing folder.
- **GetCursorPos:**
  Returns the position of the cursor.
- **GetMiletusPath:**
  Returns the common paths.
- **StartFileDrag:**
  Start dragging a file from your application to any destination where the file is accepted.

LOCAL DATABASE
ACCESS SUPPORT IN MILETUS.

There is also support for direct local database access!
The component **TMiletusClientDataSet** makes it easy for a **Miletus** application to create and use local databases by a familiar syntax of using **TClientDataSet**. It also allows a seamless integration of multiple types of databases with data-aware components like **TWebDBGrid, TWebDBTableControl, TWebDBEdit** etc...
All the database operations can be done in the standard **Delphi** way through the **TMiletusClientDataSet** component.

COMPONENTS

- **TMiletusOpenDialog:**
  Displays a native open dialog and returns the selected path(s).
- **TMiletusSaveDialog:**
  Similarly to **TMiletusOpenDialog,** it displays a native save dialog and returns the selected path.
- **TMiletusMessageBox:**
  Shows a native message dialog. The labels, the dialog type, the buttons and the verification checkbox are all customizable.
- **TMiletusErrorBox:**
  Shows a native error message dialog.
- **TMiletusMainMenu:**
  Creates and appends a native main menu to the form where it's dropped.
- **TMiletusPopupMenu:** Creates and displays a native popup menu
- **TMiletusNotificationCenter:**
  Allows you to show notifications on the operating system.
- **TMiletusWindow:**
  Allows the creation of multiple application windows which can be linked to forms or other sources.
- **TMiletusTrayIcon:**
  Creates a tray icon on the OS tray. An optional popup menu can be assigned to it.
- **TMiletusFileWatcher:**
  Monitors a list of files for changes. Each file has its own event handler which will be triggered when the file has changed.
- **TMiletusGlobalShortcuts:**
  Add a list of keyboards shortcuts that will be recognized even when the application is not in focus.

For now, there are 5 database drivers that can be used with `TMiletusClientDataSet`:

`TMiletusAccessDBDriver` for   **MS Access** databases

`TMiletusMySQLDBDriver` for   **MySQL** databases

`TMiletusSQLiteDBDriver` for   **SQLite** databases

`TMiletusPostgreSQLDBDriver` for   **PostgreSQL** databases

`TMiletusMSSQLDBDriver` for   **MS** SQL databases and more are coming…

TARGETS

At this moment, from a **Miletus** application,
 4 different target desktop application types can be created:
- Win 32bit
- Win 64bit
- macOS 64bit

A Miletus application using a TMS FNC Planner component running on a mac

TMS Web Miletus FNC NavigationPanel Demo

Treeview (5 new items !)

Appearance

2021-05-07

Settings

Hello World !

Image

Treeview (5 new items !)

Alfa Romeo	156 1,6TS	1598	88	4	120	69900(
Alfa Romeo	156 1,8TS	1774	106	4	144	76900(
Alfa Romeo	156 2,0TS	1970	114	4	155	89900(
Alfa Romeo	156 2,5	2492	140	6	190	109990(
Alfa Romeo	166 2,0TS	1970	114	4	155	110000(
Alfa Romeo	166 2,0V6	1996	151	6	190	135000(
Alfa Romeo	166 2,5V6	2492	140	6	190	146000(
Alfa Romeo	166 3,0V6	2959	166	6	226	158000(
Alfa Romeo	Spider 1,8	1747	106	4	144	99900(
Alfa Romeo	Spider 2,0	1970	114	4	155	107200(
Alfa Romeo	Spider 3,0	2959	141	6	192	143700(
Audi	A3 1,6	1595	74	4	101	69000(
Audi	A3 1,8	1781	92	4	125	76400(
Audi	A3 1,9TDI	1896	66	4	90	89000(
Audi	A4 1,6	1595	74	4	101	83500(
Audi	A4 1,8	1781	92	4	125	93300(
Audi	A4 2,4	2393	120	6	163	106500(
Audi	A4 2,8	2771	142	6	193	131350(
Audi	A4 1,9TDI	1896	66	4	90	89700(
Audi	TT Coupe	1781	132	4	180	123500(
Audi	A6 1,8	1781	92	4	125	109300(

A Miletus application also with numerous TMS FNC UI components running on Linux directly

These different targets can be selected from the IDE and a nice feature of Miletus technology is that it is not needed to have a macOS machine or a Linux machine to create the necessary application files to deploy to these machines when developing with Delphi on Windows. At the same time, the compiler can generate a Miletus Windows application from a macOS or Linux machine etc…

WHY CHOOSE MILETUS?

Miletus is not meant to be an Electron replacement, but rather something to co-exists next to the TMS WEB Core current Electron support as an alternative to those who want:

- **No NodeJS dependency** (used with Electron)
- **Smaller executable sizes**
- **Less deployed files** (no browser to deploy like with Electron)
- **More broad local database support**

You can take full advantage of web technologies combined with native operating system level operations. There's a lot of native functionality exposed already. Access to the local file system, operating system menus and notifications, drag and drop functionality, global shortcuts just to name a few.

You don't necessarily need a DB backend running, you can easily connect to a local DB file or a cloud DB service just like in a VCL or FireMonkey application!
For cross platform targets, the following databases are supported: SQLite, MSSQL, PostgreSQL, mySQL with more on the list and coming.
The power of **HTML5** and **CSS3** is at your disposal.
There is a huge amount of libraries and templates available for web applications that you can not only reuse in your Miletus application but with their help you can also create visually more pleasing user interfaces!
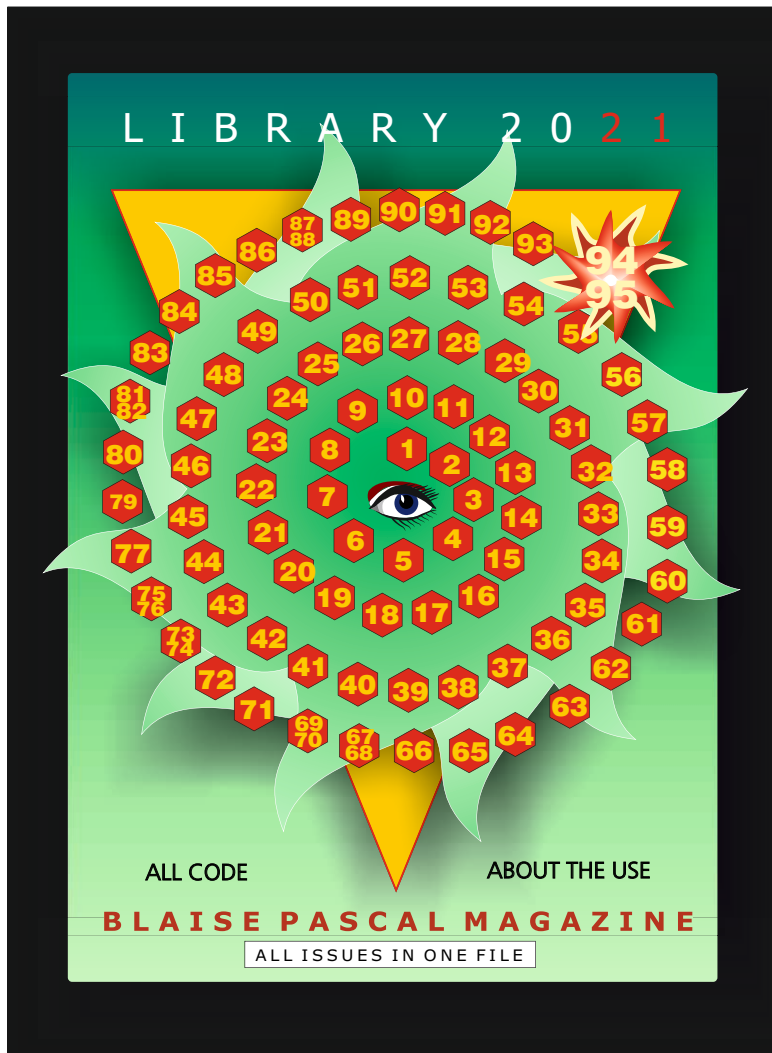
MILETUS

By Boian Mitov

starter ——————— expert

INTRODUCTION

Since the early ages of computing, two competing types of languages have emerged. Static languages, and Dynamic languages, each with its strengths and weaknesses.

Although there is a fair number of exceptions - dynamic languages are usually implemented as interpreters - and typically used for scripting, where static languages are implemented as compilers, and are usually used for developing complex application software.

There is a very good reason for this.
In static languages such as **Delph**i and **C++**, types are in general defined in declarations and the language enforces that whenever they are used the code will match the definition as declared. As example in Delphi we declare a class, and in the declaration we declare all the fields, properties and methods.
If you declare a class `TClass1` that has only one method – `Method1( AValue : Integer );`, and you create instance of the class called `AObject1`, then you can call `Method1` with `value` of `integer`, but you can't call `Method2` or pass other types of data who are incompatible with `integer`. The compiler will not be able to compile your code, and will show you a compile time error.
This is not the case with most Dynamic languages such as **Java Script**, and **Python.**

From te editor: The last is very popular and it is good to know that its origins are Dutch and created in Delphi. I have even met the originator since he was invited by our usergroup.

JavaScript often abbreviated as JS, is a programming language that conforms to the ECMAScript specification. JavaScript is high-level, often just-in-time compiled, and multi-paradigm. It has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions.
As a multi-paradigm language - (computing, of a programming language) Supporting more than one programming paradigm, in order to allow the most suitable programming style for a task.
JavaScript engines were originally used only in web browsers, but they are now core components of other software systems, most notably servers and a variety of applications.
Python is an interpreted high-level general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.

Guido van Rossum began working on Python in the late 1980s, as a successor to the ABC programming language.

In most modern dynamic languages, you can create `objects,` and then add or remove `methods,` and fields to them during the execution. It is a common practice to start with an empty object and then start adding all the `methods` and `fields` that you need to it.
As example you can create an object called `Object1` that has no methods.
Later on during the execution you can add a method called `Method1.` From that point on you can call the method on the object.
At some point you can decide to add some more methods, and so on.

As you can see the two types of languages follow completely different basic philosophy, that directly leads to their corresponding advantages and disadvantages.

With static languages, both you and the compiler know what object of certain type can and can't do anywhere in the code.

As example if a `function` accepts an object of type `TClass1` as parameter `AValue1,` the function knows that `AValue1` is `TClass1` and it only can execute `AValue1.Method1` with integer compatible parameter. Anything else will generate compiler error.

DYNAMIC LANGUAGES.

This is not the case with Dynamic languages. No one knows what capabilities the variable a function receives will have. It may or may not have certain methods and or fields. Even if you call the same function with the same object at different times the object is not guaranteed to still have the same functionality as before.

The dynamic nature of the languages such as `JavaScript` and `Python` is what makes them attractive for quick and dirty prototyping, and for scripting purposes. This is also what makes them a nightmare when developing complex frameworks.

During the development any bug introduced will be discovered only when the affected code will be executed days or sometimes months or years down the road. To mitigate this strong typed more static super sets of some popular scripting languages have started to appear such as `TypeScript.`

One of the most powerful advantages of Dynamic languages however is their ability for adding brand new types of objects during execution. Effectively an end user can design and start using their own object customized to his/hers needs. While this is not something that is often needed, there are many cases where such functionality is highly desirable.

When `Delphi` was introduced 26 years ago, it was an absolutely revolutionary product! Here was a powerful static strong typed compiler Object Oriented language, that in addition introduced easy to use visual development.

TypeScript is a programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language. TypeScript is designed for the development of large applications and transcompiles to JavaScript. As TypeScript is a superset of JavaScript, existing JavaScript programs are also valid TypeScript programs. It may be used to develop JavaScript applications for both client-side and server-side execution.

There are multiple options available for transcompilation. Either the default TypeScript Checker can be used, or the Babel compiler can be invoked to convert TypeScript to JavaScript. TypeScript supports definition files that can contain type information of existing JavaScript libraries.

There are third-party header files for popular libraries such as jQuery, MongoDB, and D3.js. TypeScript headers for the Node.js basic modules are also available, allowing development of Node.js programs within TypeScript.

The TypeScript compiler is itself written in TypeScript and compiled to JavaScript. Anders Hejlsberg, lead architect of C# and creator of Delphi and Turbo Pascal, has worked on the development of TypeScript.

There was however something that in the excitement most people missed. Under the hood there was a hidden gem called `RTTI – Run Time Type Info`. Yes, Delphi 26 years ago was one of the pioneers in the `RTTI` revolution.

With the `RTTI` at that time it was possible to query an object and obtain the list of all of its published properties, and methods.

Although this `RTTI` implementation was relatively limited, it allowed developers to be able to easily save and restore objects to and from disk, or from communication with remote machine, or to allow users to edit the properties of existing objects from the user interface. Since then Delphi however continued to evolve and gain even more power. Over the years, Operator overloading was added, then `Generics,` followed by support for functional programming with `Anonymous` method, and finally really modern

and very advanced **RTTI,** with the nice touch of support for custom attributes.

RTTI

The new **DELPHI RTTI** is extremely powerful. With it you can query all the units in your project, all the types in the units, and all of their properties, methods, and fields. You can read and write values in properties and fields, and even execute methods. All this was great, however unfortunately the new **RTTI** came with somewhat cumbersome, difficult to use, and error prone **API.**

Since the new **RTTI** functionality was essential for the further development of all of the component libraries I was developing at the time, I decided to spend fair amount of time, and to develop easier to use and more powerful **API** around the advanced **RTTI.**

Thus the **Mitov.TypeInfo** was born. It is part of the free **Mitov.Runtime** library, and I already published article about it in Issue 47 - Nr 9 2015 BLAISE PASCAL MAGAZINE "**THE ENHANCED RTTI (RUNTIME TYPE LIBRARY)**" . Over the following years, I redesigned all of the Mitov Software libraries to heavily utilize and benefit from the new **RTTI,** significantly simplifying and reducing all the code, while also introducing huge number of new features. Based on the new **RTTI,** I also developed the now widely used by the libraries Visual Live Bindings technology.

Since I was also working on **OpenWire Studio** – a still in Beta **OpenWire** based development environment, I designed the entire **OpenWire** studio based on the new **RTTI** as well.

When 6 years ago I started working with **Arduino** micro controllers, I decided to create a special version of **OpenWire** Studio designed to program **Arduino,** and later named this version **Visuino.**

Since then, the **Visuino** grew and became by far the most popular **Mitov Software** product. Originally **Visuino** was envisioned as a simple easy to use development environment for Makers, Students, Artists, and Hardware Engineers to be able to program **Arduino** projects.

Very soon however it proved to be extremely powerful tool, surprising even me by how easy it was to create very complex and powerful projects.

As I kept developing it, I improved the optimization of the generated code to the point where **Visuino** started generating more efficient **Arduino** code that even what I was able to write manually.
This resulted in people developing projects of never seen before complexity in the **Arduino** world, and resulting in very complex **OpenWire** diagrams, sometimes with hundreds of components. At this point, the need to be able to split big projects into sub projects became apparent, and I started working on a professional version of **Visuino** supporting sub diagrams that can be used as components in each other or in the main diagram.

Once a sub diagram is created, a new component has to appear in the component toolbar of the rest of the diagrams allowing instances of the sub diagram to be added to the other diagrams.
This presents a challenge.
The sub diagram effectively has to appear as a brand new component class that has all the pins and properties as added to the sub diagram.
Since all visual components in **Visuino** are **VCL** components, accessed through the **RTTI** in order to be drawn in the diagram and edited in the **Object Inspector**, I needed a way to create this new virtual component class and add it to the **RTTI.**

Fortunately from the beginning, I had designed the new **RTTI API** to be open expandable architecture. This allowed me to extend it to allow the creation of virtual class types, and adding virtual property info in them.
This was the birth of a brand new technology based on the **RTTI,** allowing the **RTTI** to be dynamically expanded during execution, and allowing new Delphi types to be created, removed and modified at runtime.

I called this new technology
Dynamic Type Info – DTI.

The initial **DTI** implementation was relatively limited, but it quickly proved itself, and made **Visuino Pro** a real success.

In short time it became a popular tool for even very experienced developers using **Arduino** and similar micro-controllers.

As **Visuino** grew in popularity, I started to receive increasing number of requests for adding support for more and more micro controllers, shields, sensors, and actuators.

Although I have made it extremely easy to create new components for **Visuino** using **Delphi,** I quickly became overwhelmed by the sheer number of requests.

To resolve the challenge I created the **Visuino Delphi SDK.** Using **Delphi,** with the SDK, anyone could create new components with minimal effort.

This worked fine and a few people started developing **Visuino** components, however since the components were compiled into packages that were loaded by **Visuino,** every time I released a new version of **Visuino,** the packages had to be recompiled and everyone needed to release updated version.

Furthermore very few people were experienced enough in **Delphi** development to be able to create the component packages.

This was obviously not an acceptable solution, and a new way to develop components for **Visuino** was needed.

OBVIOUS CHOICE

Once again the DTI was the obvious choice. Instead of having the **Visuino** components compiled into packages, I can simply define them in plain text files, and have **Visuino** parse the text files, and create the virtual component classes in **DTI.**

It took 6 months of heavy work, but at the end I had the new **Visuino** design working, and most of the **Visuino** components converted to plain text format.

I also made all the of text format components open source, so anyone can study, modify and expand them.

To achieve all this I expanded and improved the **DTI** considerably.

It is now fully feature complete, and has become a very powerful technology allowing Delphi to compete with any modern dynamic language.

As I have done with the **Mitov.RTTI,** I added the new **DTI** to **Mitov.Runtime** and made it free download allowing anyone to benefit from the new functionality.

Now that I have introduced you to the history of the **DTI** development, it is time to show you with a simple project how you can use it in your own code.

We will create a couple of virtual classes, add properties to them, and will create object instances of the virtual dynamic classes.

For this, we will create a simple **Delphi VCL** application, and add **TMemo** on the form so we can use it to print reports on how the types and the object instances change during the execution.

The following code has several visual aspects:
The standard code will be in Courier colored.
The special additions like background color
under the code shows that it needs to be
added for example.

Next we will add few units that we will use in
the code:

```
uses
 …
 System.TypInfo, System.Rtti, Mitov.TypeInfo, Mitov.Containers.List,
 Mitov.Containers.Dictionary, Mitov.Utils;
```

The **System.TypInfo, System.Rtti** provide
Delphi types that are needed for the **RTTI**.
Mitov.TypeInfo contains the **Mitov RTTI**
including the new **DTI** support.
The rest of the units contain containers and
utils that we will use in the demo code.

To help us observe the changes in the dynamic
types and the object instances, we will first
prepare two debug printing functions:

```
procedure TForm1.ReportTypeInfo( const ATypeName : String );
begin
 var ATypeInfo : ITypeInfo;
 if( TRttiInfo.GetType( ATypeName, ATypeInfo )) then
  begin
  Memo1.Lines.Add( '----- Type -----' );
  Memo1.Lines.Add( ATypeInfo.Name );
  for var APoperty in ATypeInfo.SingleProperties do
   Memo1.Lines.Add( APoperty.Name + ' : '
        + APoperty.TypeInfo.Name + ' = '
        + APoperty.Default[ NIL ].AsString() );
   Memo1.Lines.Add( '---------------' );
   Memo1.Lines.Add( '' );
  end;
end;
```

and

```
procedure TForm1.ReportInstance( AInstance : TObject );
begin
 Memo1.Lines.Add( '--- Instance ---' );
 var ATypeInfo := AInstance.TypeInfo();
 for var AProperty in ATypeInfo.SingleProperties do
  Memo1.Lines.Add( ATypeInfo.Name + '.'
       + AProperty.Name
       + ' = ' + AProperty.Value[ AInstance ].ToString() );

 Memo1.Lines.Add( '---------------' );
 Memo1.Lines.Add( '' );
end;
```

The ReportTypeInfo function will list the
properties of any **RTTI** type by name.
The ReportInstance function will use the **RTTI**
to get the list of all the of object instance's
single properties and their values.
To hold the information for our dynamic types
we will create a list of objects in the form:

```
TForm1 = class(TForm)
 …
 private
 FDynamicClasses : IArrayList<IDynamicTypeInfo>;
 …
```

We can initialize the **FDynamicClasses**
in the Form's **OnCreate** event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 FDynamicClasses := TArrayList<IDynamicTypeInfo>.Create();
 …
```

Now we will create the class for the objects
containing the dynamic types. For simplicity
we will only create class types, but with **DTI**
you can create practically any dynamic type –
records, interfaces, enumerated types or
anything else:

```
TMyDynamicClassDefinition = class( TInterfacedObject, IDynamicTypeInfo )
 protected
  FName     : String;
  FProperties : IArrayList<IMyDynamicPropertyDefinition>;

 protected
  function GetName(): String;
  function GetDeclaredSingleProperties(): ISinglePropertiesInfo;

 protected // IDynamicTypeInfo
  function GetTypeInfo( const ATypeInfo : ITypeInfo ): ITypeInfo;
  procedure Populate( AOwnerObject : Tobject;
        const AMember : IValueMemberInfo; AObject : TObject; ARootInstance : TPersistent );

 public
  function CreateInstance(): TMyDynamicObjectInstance;

 protected
  procedure DestroyingInstance( AInstance : TMyDynamicObjectInstance );

 public
  property Properties : IArrayList<IMyDynamicPropertyDefinition> read FProperties;

 public
  constructor Create( const AName : String );

 end;
```

The class is interfaced so it will be automatically destroyed. It has to implement `IDynamicTypeInfo`. This interface allows the class to provide its own dynamic type info to the **RTTI.** In the class first we have the `FName` that will hold the name of the dynamic type. The `FName` is assigned in the constructor.

```
constructor TMyDynamicClassDefinition.Create( const AName : String );
begin
  inherited Create();
  FName := AName;
  Fproperties := TArrayList<IMyDynamicPropertyDefinition>.Create();
end;
```

The `FProperties` list will contain the definitions of the dynamic properties. Here we can add or remove properties. The class also has a property that allows access to the properties list:

```
property Properties : IArrayList<IMyDynamicPropertyDefinition> read Fproperties;
```

We will look into `IMyDynamicPropertyDefinition` after we finish with the `TMyDynamicClassDefinition`. The `GetName()` method allows public access to the `FName` field:

```
function TMyDynamicClassDefinition.GetName(): String;
begin
 Result := FName;
end;
```

The `GetTypeInfo` and Populate are the methods implementing `IDynamicTypeInfo`. They are the only two methods that any **Dynamic Type Info** definition class has to implement. `GetTypeInfo` will return the dynamic type info:

```
function TMyDynamicClassDefinition.GetTypeInfo( const ATypeInfo : ITypeInfo ): ITypeInfo;
begin
 Result := TMyDynamicTypeInfo.Create( Self );
end;
```

The Dynamic Type Info is implemented in **TMyDynamicTypeInfo**. We will look into this class a bit later. The **Populate method** is a bit more advanced and allows creating inheritance of one dynamic type from another. For simplicity, we will leave it as empty implementation in this project.

```
procedure TMyDynamicClassDefinition.Populate( AOwnerObject : TObject; const AMember : IValueMemberInfo;
                                              AObject : TObject; ARootInstance : TPersistent );
begin
end;
```

The **GetDeclaredSingleProperties()** will create and return **ISinglePropertiesInfo** – a list of **ISinglePropertyInfo**:

```
function TMyDynamicClassDefinition.GetDeclaredSingleProperties() : ISinglePropertiesInfo;
begin
 Result := TDynamicSinglePropertiesInfo.Create();
 for var AProperty in FProperties do
  Result.Add( AProperty.GetPropertyInfo() );

end;
```

CreateInstance allows creating object instances of the dynamic type, and will call **RegisterInstance** for each property so virtual property instance can be created for the object:

```
function TMyDynamicClassDefinition.CreateInstance() : TMyDynamicObjectInstance;
begin
 Result := TMyDynamicObjectInstance.Create( Self );
 for var AProperty in FProperties do
  AProperty.RegisterInstance( Result );
end;
```

We will look into the **TMyDynamicObjectInstance** class later.
DestroyingInstance is called when destroying the objects. It is used by the **TMyDynamicObjectInstance** destructor as you will see later. It will simply call **UnregisterInstance** for each property so the virtual property instances for the object can be destroyed:

```
procedure TMyDynamicClassDefinition.DestroyingInstance( AInstance : TMyDynamicObjectInstance );
begin
 for var AProperty in FProperties do
  AProperty.UnregisterInstance( AInstance );

end;
```

Now we will declare our **TMyDynamicTypeInfo** class. This class will implement the **RTTI type** info for the dynamic type definition:

```
TMyDynamicTypeInfo = class( TDynamicClassTypeInfo )
 protected
  FOwner : TMyDynamicClassDefinition;

 protected
  function GetName() : String; override;
  function GetHandle() : PTypeInfo; override;

 protected // ITypeInfo
  function GetDeclaredSingleProperties() : ISinglePropertiesInfo; overload; override;

 protected // IClassTypeInfo
  function GetMetaclassType() : TClass; override;

 public
  constructor Create( AOwner : TMyDynamicClassDefinition );

 end;
```

The class extends the `TDynamicClassTypeInfo` defined in `Mitov.TypeInfo`. It has `Fowner` pointer to the `TMyDynamicClassDefinition`. The constructor simply assigns the `FOwner`:

```
constructor TMyDynamicTypeInfo.Create( AOwner : TMyDynamicClassDefinition );
begin
 inherited Create();
 FOwner := AOwner;
end;
```

The `GetHandle` and `GetMetaclassType` will be implemented to return the `TDynamicClassTypeInfo` type info:

```
function TMyDynamicTypeInfo.GetHandle() : PTypeInfo;
begin
 Result := System.TypeInfo( TMyDynamicClassDefinition );
end;

function TMyDynamicTypeInfo.GetMetaclassType() : TClass;
begin
 Result := TMyDynamicClassDefinition;
end;
```

And finally `GetName` and `GetDeclaredSingleProperties` simply will call the corresponding methods of the `TMyDynamicClassDefinition` object:

```
function TMyDynamicTypeInfo.GetName() : String;
begin
 Result := FOwner.GetName();
end;

function TMyDynamicTypeInfo.GetDeclaredSingleProperties() : ISinglePropertiesInfo;
begin
 Result := FOwner.GetDeclaredSingleProperties();
end;
```

The support for the dynamic class type is completed. Now it is time to add the support for properties. To simplify the memory management I have decided to implement the property class as interfaced object. This is purely optional, and you can implement it any way you want. Here is the interface definition:

```
IMyDynamicPropertyDefinition = interface
 ['{396DF71C-3AAF-4DB3-B3E1-967545E269C9}']

 function GetPropertyInfo() : ISinglePropertyInfo;
 procedure RegisterInstance( AInstance : TMyDynamicObjectInstance );
 procedure UnregisterInstance( AInstance : TMyDynamicObjectInstance );
end;
```

Here is the class definition. To simplify the demo I have implemented only support for `String` type properties, but it is very easy to add support for any other type:

```
TMyDynamicPropertyDefinition = class( TInterfacedObject, IMyDynamicPropertyDefinition )
protected
 FName    : String;
 FDefault : String;
 FValues  : IDictionary<Pointer, String>;

protected // IMyDynamicPropertyDefinition
 function GetPropertyInfo() : ISinglePropertyInfo;
 procedure RegisterInstance( AInstance : TMyDynamicObjectInstance );
 procedure UnregisterInstance( AInstance : TMyDynamicObjectInstance );

public
 function GetName() : String;
 function GetDefault( AInstance : Pointer ) : TValue;

 function GetValue( AInstance : Pointer ) : TValue;
 procedure SetValue( AInstance : Pointer; const AValue : TValue );

public
 constructor Create( const AName : String; const ADefaultValue : String );
end;
```

The **Fname** field will store the name of the property.

The **FDefault** field will hold the default value for the property.

Since each instance of a virtual dynamic object will need instances of the dynamic properties, we can use **FValues** dictionary to store the current value for each instance property associated with the owner object.

```
FValues  : IDictionary<Pointer, String>;
```

The **dictionary** will use the pointer to the object as key and will hold the String value of the associated property.

RegisterInstance will add and assign default to the values of the dictionary when new object instance is created:

```
procedure TMyDynamicPropertyDefinition.RegisterInstance( AInstance : TMyDynamicObjectInstance );
begin
 FValues[ AInstance ]:= FDefault;
end;
```

UnregisterInstance will remove values from the dictionary when object instance is deleted:

```
procedure TMyDynamicPropertyDefinition.UnregisterInstance( AInstance :
TMyDynamicObjectInstance );
begin
 FValues.Remove( AInstance );
end;
```

GetValue and SetValue will get and set values for instance in the dictionary:

```
function TMyDynamicPropertyDefinition.GetValue( AInstance : Pointer ) : TValue;
begin
 Result := FValues[ AInstance ];
end;

procedure TMyDynamicPropertyDefinition.SetValue( AInstance : Pointer; const AValue : TValue );
begin
 FValues[ AInstance ]:= AValue.AsString();
end;
```

The GetName and GetDefault methods simply return the corresponding fields:

```
function TMyDynamicPropertyDefinition.GetName() : String;
begin
 Result := FName;
end;

function TMyDynamicPropertyDefinition.GetDefault( AInstance : Pointer ) : TValue;
begin
 Result := FDefault;
end;
```

GetPropertyInfo creates and returns the property type info:

```
function TMyDynamicPropertyDefinition.GetPropertyInfo() : ISinglePropertyInfo;
begin
 Result := TMyDynamicPropertyInfo.Create( Self );
end;
```

We will look into `TMyDynamicPropertyInfo` next. It will contain the `RTTI` info for the dynamic property:

```
TMyDynamicPropertyInfo = class( TDynamicSinglePropertyInfo )
protected
 FOwner : TMyDynamicPropertyDefinition;

protected // INamedObjectInfo
 function GetName() : String; override;

protected // ITypedObjectInfo
 function GetTypeInfo() : ITypeInfo; override;

protected // IValueMemberInfo
 function GetDefault( AInstance : Pointer ) : TValue; override;

 function GetValue( AInstance : Pointer ) : TValue; override;
 procedure SetValue( AInstance : Pointer; const AValue : TValue ); override;

public
 constructor Create( AOwner : TMyDynamicPropertyDefinition );

end;
```

The class inherits from `TDynamicSinglePropertyInfo` (defined in `Mitov.TypeInfo`), and is very simple. It has a `FOwner` pointer to the `TMyDynamicPropertyDefinition` assigned in the constructor:

```
constructor TMyDynamicPropertyInfo.Create( AOwner : TMyDynamicPropertyDefinition );
begin
 inherited Create();
 FOwner := AOwner;
end;
```

`GetTypeInfo` returns the type info for `String`:

```
function TMyDynamicPropertyInfo.GetTypeInfo() : ITypeInfo;
begin
 Result := TRttiInfo.TypeOf<String>();
end;
```

Finally the `GetName, GetDefault, GetValue,` and `SetValue` are simply calling the corresponding methods in `TMyDynamicPropertyDefinition`:

```
function TMyDynamicPropertyInfo.GetName() : String;
begin
 Result := FOwner.GetName();
end;

function TMyDynamicPropertyInfo.GetDefault( AInstance : Pointer ) : TValue;
begin
 Result := FOwner.GetDefault( AInstance );
end;

function TMyDynamicPropertyInfo.GetValue( AInstance : Pointer ) : TValue;
begin
 Result := FOwner.GetValue( AInstance );
end;

procedure TMyDynamicPropertyInfo.SetValue( AInstance : Pointer; const AValue :
TValue );
begin
 FOwner.SetValue( AInstance, AValue );
end;
```

Once we have defined the new dynamic type, we should be able to create object instances of it. To represent the object instance of the virtual object, we need to define an Object Instance class. We already looked at the `TMyDynamicClassDefinition.CreateInstance` method that returns the virtual object instance, and here is the definition of the `TMyDynamicObjectInstance` class:

```pascal
TMyDynamicObjectInstance = class( TInterfacedPersistent, IDynamicTypeInfo )
protected
 FDefinition : TMyDynamicClassDefinition;

protected // IDynamicTypeInfo
 function GetTypeInfo( const ATypeInfo : ITypeInfo ) : ITypeInfo;
 procedure Populate( AOwnerObject : Tobject;
            const AMember : IValueMemberInfo; AObject : TObject; ARootInstance : TPersistent );

protected
 constructor Create( ADefinition : TMyDynamicClassDefinition );
 destructor Destroy(); override;

end;
```

It contains a **FDefinition** field pointer to the **TMyDynamicClassDefinition** that creates it. The field is initialized in the constructor:

```pascal
constructor TMyDynamicObjectInstance.Create( ADefinition
 : TMyDynamicClassDefinition );
begin
 inherited Create();
 FDefinition := ADefinition;
end;
```

The destructor will simply inform the **FDefinition** that the object is being destroyed by calling **DestroyingInstance,** so any virtual property instances can be removed from the property dictionaries:

```pascal
destructor TMyDynamicObjectInstance.Destroy();
begin
 FDefinition.DestroyingInstance( Self );
 inherited;
end;
```

Finally **GetTypeInfo** and **Populate** will simply call the corresponding methods of **FDefinition**:

```pascal
function TMyDynamicObjectInstance.GetTypeInfo( const ATypeInfo : ITypeInfo ) : ITypeInfo;
begin
 Result := FDefinition.GetTypeInfo( ATypeInfo );
end;

procedure TMyDynamicObjectInstance.Populate( AOwnerObject : Tobject;
        const AMember : IValueMemberInfo; AObject : TObject; ARootInstance : TPersistent );
begin
 FDefinition.Populate( AOwnerObject, AMember, AObject, ARootInstance );
end;
```

We are done with the implementation of the **DTI** support for our project.
Now it is time to test it by creating our first dynamic type. We will call it **'TMyVirtualClass'**.
For simplicity in my project I will do this in the **TForm1.FormCreate**:

```pascal
procedure TForm1.FormCreate(Sender: TObject);
begin
 FDynamicClasses := TArrayList<IDynamicTypeInfo>.Create();

 var AVirtualClass1 := TMyDynamicClassDefinition.Create( 'TMyVirtualClass' );
AVirtualClass1.Properties.Add( TMyDynamicPropertyDefinition.Create( 'MyProperty', 'Hello' ) );
FDynamicClasses.Add( AVirtualClass1 );
```

Next we will create and add a virtual property definition. We will name the property 'MyProperty' and will assign it default value of 'Hello':
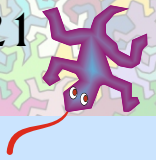
```
AVirtualClass1.Properties.Add(
TMyDynamicPropertyDefinition.Create( 'MyProperty', 'Hello' ) );
```

We will add our new type definition to our type definitions list:

```
FDynamicClasses.Add( AVirtualClass1 );
```

Finally we have to register our dynamic type to the RTTI:

```
TRttiInfo.RegisterType( AVirtualClass1.TypeInfo() );
```

Here we call the TypeInfo() class helpers method to obtain the dynamic type info and register it with the TRttiInfo.RegisterType method.

Before we shutdown the application we should unregister any dynamic types. We can do this in the forms OnDestroy event handler by calling TRttiInfo.UnregisterType for each type:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
 for var AItem in FDynamicClasses do
  TRttiInfo.UnregisterType( ( AItem as TObject ).TypeInfo() );

end;
```

Now we can test our new type definition. If we execute:

```
for var AType in TRttiInfo.Types do
 Memo1.Lines.Add( AType.Name );
```

It will list all types from the RTTI, and we should be able to see our type in the list:

```
...
TMessageManager.TListenerWithId
TMessageManager.TListenerList
TMessageManager
TMyVirtualClass
```

We can also call:

```
ReportTypeInfo( 'TMyVirtualClass' );
```

and it will display:

```
 ----- Type -----
TMyVirtualClass
MyProperty : string = Hello
 ----------------
```

Next we can create one or more instances of the new dynamic type:

```
var AInstance1 := AVirtualClass1.CreateInstance();
```

and we can use **ReportInstance** to see the property values of our instance:

```
ReportInstance( AInstance1 );
```

it should display:

```
--- Instance ---
TMyVirtualClass.MyProperty = Hello
 ----------------
```

Now we can use the RTTI to assign new value to the MyProperty property of Ainstance1:

```
var APropertyInfo : ISinglePropertyInfo;
if( AInstance1.TypeInfo().GetSingleProperty( 'MyProperty', APropertyInfo )) then
 APropertyInfo.Value[ AInstance1 ] := 'World';
```

If we call the **ReportInstance** again:

```
ReportInstance( AInstance1 );
```

It should display:

```
--- Instance ---
TMyVirtualClass.MyProperty = World
----------------
```

We have fully functional dynamic type and instance of it. To do some more experimenting we can create a bit more complex type and modify it at runtime. We will create a **'TDeveloper'** class with two properties – **FirstName** and **LastName**. I will also assign some default values to the properties:

```
var AVirtualClass2 := TMyDynamicClassDefinition.Create( 'TDeveloper' );
AVirtualClass2.Properties.Add( TMyDynamicPropertyDefinition.Create( 'FirstName', 'Boian' ));
AVirtualClass2.Properties.Add( TMyDynamicPropertyDefinition.Create( 'LastName', 'Mitov' ));
FDynamicClasses.Add( AVirtualClass2 );
TRttiInfo.RegisterType( AVirtualClass2.TypeInfo() );
```

If we call:

```
ReportTypeInfo( 'TDeveloper' );
```

it will report:

```
----- Type -----
TDeveloper
FirstName : string = Boian
LastName : string = Mitov
```

After this at some point we can expand the type by adding another property:

```
AVirtualClass2.Properties.Add( TMyDynamicPropertyDefinition.Create( 'Language', 'Delphi' ));
```

Calling:

```
ReportTypeInfo( 'TDeveloper' );
```

will report:

```
----- Type -----
TDeveloper
FirstName : string = Boian
LastName : string = Mitov
Language : string = Delphi
----------------
```

Again once created we can change the values of the instance properties:

```
if( AInstance2.TypeInfo().GetSingleProperty( 'Language', APropertyInfo )) then
  APropertyInfo.Value[ AInstance2 ] := 'Object Pascal';
```

Calling:

```
ReportInstance( AInstance2 );
```

We should see:

```
--- Instance ---
TDeveloper.FirstName = Boian
TDeveloper.LastName = Mitov
TDeveloper.Language = Object Pascal
----------------
```

As you can see, it is fairly easy to add Dynamic Type Info functionality to your Delphi application, and gain the same advantages that the developers of Dynamic languages enjoy. You can use the functionality to create flexible configurable and user expandable applications and more. For me it made it possible for complete beginners to be able to start making Visuino components with just a text editor. It also allowed me introduce hierarchical Visuino diagrams. The technology however can be used for many other purposes. From flexible scripting, to customizable user interface, and more...Your Imagination is the only limit.

Here is the complete source code for the project that I created while writing the article:

```pascal
unit Unit1;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls, System.TypInfo, System.Rtti, Mitov.TypeInfo,
  Mitov.Containers.List, Mitov.Containers.Dictionary, Mitov.Utils;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    FDynamicClasses : IArrayList<IDynamicTypeInfo>;

  private
    procedure ReportTypeInfo( const ATypeName : String );
    procedure ReportInstance( AInstance : TObject );

  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

type
  TMyDynamicClassDefinition = class;
  TMyDynamicObjectInstance = class;
//---------------------------------------------------------------------
  IMyDynamicPropertyDefinition = interface
    ['{396DF71C-3AAF-4DB3-B3E1-967545E269C9}']

    function GetPropertyInfo(): ISinglePropertyInfo;
    procedure RegisterInstance( AInstance : TMyDynamicObjectInstance );
    procedure UnregisterInstance( AInstance : TMyDynamicObjectInstance );

  end;
//---------------------------------------------------------------------
  TMyDynamicPropertyDefinition = class( TInterfacedObject, IMyDynamicPropertyDefinition )
  protected
    FName    : String;
    FDefault : String;
    FValues  : IDictionary<Pointer, String>;

  protected // IMyDynamicPropertyDefinition
    function GetPropertyInfo(): ISinglePropertyInfo;
    procedure RegisterInstance( AInstance : TMyDynamicObjectInstance );
    procedure UnregisterInstance( AInstance : TMyDynamicObjectInstance );

  public
    function GetName(): String;
    function GetDefault( AInstance : Pointer ) : TValue;

    function GetValue( AInstance : Pointer ) : TValue;
    procedure SetValue( AInstance : Pointer; const AValue : TValue );

  public
    constructor Create( const AName : String; const ADefaultValue : String );

  end;
//---------------------------------------------------------------------
```

```pascal
TMyDynamicObjectInstance = class( TInterfacedPersistent, IDynamicTypeInfo )
 protected
  FDefinition : TMyDynamicClassDefinition;

 protected // IDynamicTypeInfo
  function GetTypeInfo( const ATypeInfo : ITypeInfo ) : ITypeInfo;
  procedure Populate( AOwnerObject : Tobject;
        const AMember : IValueMemberInfo; AObject : TObject; ARootInstance : TPersistent );

 protected
  constructor Create( ADefinition : TMyDynamicClassDefinition );
  destructor Destroy(); override;
 end;
//----------------------------------------------------------------------
TMyDynamicClassDefinition = class( TInterfacedObject, IDynamicTypeInfo )
 protected
  FName      : String;
  FProperties : IArrayList<IMyDynamicPropertyDefinition>;

 protected
  function GetName() : String;
  function GetDeclaredSingleProperties() : ISinglePropertiesInfo;

 protected // IDynamicTypeInfo
  function GetTypeInfo( const ATypeInfo : ITypeInfo ) : ITypeInfo;
  procedure Populate( AOwnerObject : Tobject;
        const AMember : IValueMemberInfo; AObject : TObject; ARootInstance : TPersistent );

 public
  function CreateInstance() : TMyDynamicObjectInstance;

 protected
  procedure DestroyingInstance( AInstance : TMyDynamicObjectInstance );

 public
  property Properties : IArrayList<IMyDynamicPropertyDefinition> read FProperties;

 public
  constructor Create( const AName : String );
 end;
//----------------------------------------------------------------------
TMyDynamicTypeInfo = class( TDynamicClassTypeInfo )
 protected
  FOwner : TMyDynamicClassDefinition;

 protected
  function GetName() : String; override;
  function GetHandle() : PTypeInfo; override;

 protected // ITypeInfo
  function GetDeclaredSingleProperties() : ISinglePropertiesInfo; overload; override;

 protected // IClassTypeInfo
  function GetMetaclassType() : TClass; override;

 public
  constructor Create( AOwner : TMyDynamicClassDefinition );
 end;
//----------------------------------------------------------------------
TMyDynamicPropertyInfo = class( TDynamicSinglePropertyInfo )
 protected
  FOwner : TMyDynamicPropertyDefinition;

 protected // INamedObjectInfo
  function GetName() : String; override;

 protected // ITypedObjectInfo
  function GetTypeInfo() : ITypeInfo; override;
//----------------------------------------------------------------------
```

```pascal
protected // IValueMemberInfo
  function GetDefault( AInstance : Pointer ) : TValue; override;

  function GetValue( AInstance : Pointer ) : TValue; override;
  procedure SetValue( AInstance : Pointer; const AValue : TValue ); override;

public
  constructor Create( AOwner : TMyDynamicPropertyDefinition );

end;
//--------------------------------------------------------------------
//--------------------------------------------------------------------
function TMyDynamicPropertyDefinition.GetDefault( AInstance : Pointer ) : TValue;
begin
 Result := FDefault;
end;
//--------------------------------------------------------------------
function TMyDynamicPropertyDefinition.GetValue( AInstance : Pointer ) : TValue;
begin
 Result := FValues[ AInstance ];
end;
//--------------------------------------------------------------------
procedure TMyDynamicPropertyDefinition.SetValue( AInstance : Pointer; const AValue : TValue );
begin
 FValues[ AInstance ] := AValue.AsString();
end;
//--------------------------------------------------------------------
function TMyDynamicPropertyDefinition.GetPropertyInfo() : ISinglePropertyInfo;
begin
 Result := TMyDynamicPropertyInfo.Create( Self );
end;
//--------------------------------------------------------------------
procedure TMyDynamicPropertyDefinition.RegisterInstance( AInstance : TMyDynamicObjectInstance );
begin
 FValues[ AInstance ] := FDefault;
end;
//--------------------------------------------------------------------
procedure TMyDynamicPropertyDefinition.UnregisterInstance( AInstance : TMyDynamicObjectInstance );
begin
 FValues.Remove( AInstance );
end;
//--------------------------------------------------------------------
//--------------------------------------------------------------------
constructor TMyDynamicObjectInstance.Create( ADefinition : TMyDynamicClassDefinition );
begin
 inherited Create();
 FDefinition := ADefinition;
end;
//--------------------------------------------------------------------
destructor TMyDynamicObjectInstance.Destroy();
begin
 FDefinition.DestroyingInstance( Self );
 inherited;
end;
//--------------------------------------------------------------------
function TMyDynamicObjectInstance.GetTypeInfo( const ATypeInfo : ITypeInfo ) : ITypeInfo;
begin
 Result := FDefinition.GetTypeInfo( ATypeInfo );
end;
//--------------------------------------------------------------------
procedure TMyDynamicObjectInstance.Populate( AOwnerObject : TObject; const AMember : IValueMemberInfo;
AObject : TObject; ARootInstance : TPersistent );
begin
 FDefinition.Populate( AOwnerObject, AMember, AObject, ARootInstance );
end;
//--------------------------------------------------------------------
//--------------------------------------------------------------------
```

```pascal
constructor TMyDynamicClassDefinition.Create( const AName : String );
begin
 inherited Create();
 FName := AName;
 FProperties := TArrayList<IMyDynamicPropertyDefinition>.Create();
end;
//------------------------------------------------------------------
function TMyDynamicClassDefinition.CreateInstance() : TMyDynamicObjectInstance;
begin
 Result := TMyDynamicObjectInstance.Create( Self );
 for var AProperty in FProperties do
  AProperty.RegisterInstance( Result );

end;
//------------------------------------------------------------------
procedure TMyDynamicClassDefinition.DestroyingInstance( AInstance : TMyDynamicObjectInstance );
begin
 for var AProperty in FProperties do
  AProperty.UnregisterInstance( AInstance );

end;
//------------------------------------------------------------------
function TMyDynamicClassDefinition.GetTypeInfo( const ATypeInfo : ITypeInfo ) : ITypeInfo;
begin
 Result := TMyDynamicTypeInfo.Create( Self );
end;
//------------------------------------------------------------------
procedure TMyDynamicClassDefinition.Populate( AOwnerObject : TObject; const AMember :
IValueMemberInfo; AObject : TObject; ARootInstance : TPersistent );
begin
end;
//------------------------------------------------------------------
function TMyDynamicClassDefinition.GetName() : String;
begin
 Result := FName;
end;
//------------------------------------------------------------------
function TMyDynamicClassDefinition.GetDeclaredSingleProperties() : ISinglePropertiesInfo;
begin
 Result := TDynamicSinglePropertiesInfo.Create();
 for var AProperty in FProperties do
  Result.Add( AProperty.GetPropertyInfo() );

end;
//------------------------------------------------------------------
//------------------------------------------------------------------
```

```pascal
constructor TMyDynamicTypeInfo.Create( AOwner : TMyDynamicClassDefinition );
begin
  inherited Create();
  FOwner := AOwner;
end;
//------------------------------------------------------------------
function TMyDynamicTypeInfo.GetHandle() : PTypeInfo;
begin
  Result := System.TypeInfo( TMyDynamicClassDefinition );
end;
//------------------------------------------------------------------
function TMyDynamicTypeInfo.GetMetaclassType() : TClass;
begin
  Result := TMyDynamicClassDefinition;
end;
//------------------------------------------------------------------
function TMyDynamicTypeInfo.GetName() : String;
begin
  Result := FOwner.GetName();
end;
//------------------------------------------------------------------
function TMyDynamicTypeInfo.GetDeclaredSingleProperties() : ISinglePropertiesInfo;
begin
  Result := FOwner.GetDeclaredSingleProperties();
end;
//------------------------------------------------------------------
//------------------------------------------------------------------
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.BeginUpdate();
  Memo1.Lines.Clear();

  for var AType in TRttiInfo.Types do
    Memo1.Lines.Add( AType.Name );

  Memo1.Lines.EndUpdate();
end;
//------------------------------------------------------------------
procedure TForm1.ReportTypeInfo( const ATypeName : String );
begin
  var ATypeInfo : ITypeInfo;
  if( TRttiInfo.GetType( ATypeName, ATypeInfo )) then
    begin
    Memo1.Lines.Add( '----- Type -----' );
    Memo1.Lines.Add( ATypeInfo.Name );
    for var AProperty in ATypeInfo.SingleProperties do
      Memo1.Lines.Add( AProperty.Name + ' : ' + AProperty.TypeInfo.Name + ' = ' + AProperty.Default[
NIL ].AsString() );

    Memo1.Lines.Add( '---------------' );
    Memo1.Lines.Add( '' );
    end;

end;
//------------------------------------------------------------------
procedure TForm1.ReportInstance( AInstance : TObject );
begin
  Memo1.Lines.Add( '--- Instance ---' );
  var ATypeInfo := AInstance.TypeInfo();
  for var AProperty in ATypeInfo.SingleProperties do
    Memo1.Lines.Add( ATypeInfo.Name + '.' + AProperty.Name + ' = ' + AProperty.Value[ AInstance
].ToString() );

  Memo1.Lines.Add( '---------------' );
  Memo1.Lines.Add( '' );
end;
//------------------------------------------------------------------
```

```pascal
//-----------------------------------------------------------------------
procedure TForm1.FormCreate(Sender: TObject);
begin
 FDynamicClasses := TArrayList<IDynamicTypeInfo>.Create();

 var AVirtualClass1 := TMyDynamicClassDefinition.Create( 'TMyVirtualClass' );
 AVirtualClass1.Properties.Add( TMyDynamicPropertyDefinition.Create( 'MyProperty', 'Hello' ) );
 FDynamicClasses.Add( AVirtualClass1 );

 TRttiInfo.RegisterType( AVirtualClass1.TypeInfo() );

 var AVirtualClass2 := TMyDynamicClassDefinition.Create( 'TDeveloper' );
 AVirtualClass2.Properties.Add( TMyDynamicPropertyDefinition.Create( 'FirstName', 'Boian' ));
 AVirtualClass2.Properties.Add( TMyDynamicPropertyDefinition.Create( 'LastName', 'Mitov' ));
 FDynamicClasses.Add( AVirtualClass2 );

 TRttiInfo.RegisterType( AVirtualClass2.TypeInfo() );

 ReportTypeInfo( 'TMyVirtualClass' );
 ReportTypeInfo( 'TDeveloper' );

 AVirtualClass2.Properties.Add( TMyDynamicPropertyDefinition.Create( 'Language', 'Delphi' ));
 ReportTypeInfo( 'TDeveloper' );

 var AInstance1 := AVirtualClass1.CreateInstance();

 ReportInstance( AInstance1 );

 var APropertyInfo : ISinglePropertyInfo;
 if( AInstance1.TypeInfo().GetSingleProperty( 'MyProperty', APropertyInfo )) then
  APropertyInfo.Value[ AInstance1 ] := 'World';

 ReportInstance( AInstance1 );

 var AInstance2 := AVirtualClass2.CreateInstance();

 ReportInstance( AInstance2 );

 if( AInstance2.TypeInfo().GetSingleProperty( 'Language', APropertyInfo )) then
  APropertyInfo.Value[ AInstance2 ] := 'Object Pascal';

 ReportInstance( AInstance2 );

 AInstance1.DisposeOf();
 AInstance2.DisposeOf();
end;
//-----------------------------------------------------------------------
procedure TForm1.FormDestroy(Sender: TObject);
begin
 for var AItem in FDynamicClasses do
  TRttiInfo.UnregisterType( ( AItem as TObject ).TypeInfo() );

end;
//-----------------------------------------------------------------------
end.
```

I hope you have found this article useful.
You can download the free version of **Mitov.Runtime** containing the new **Dynamic Type Info**
functionality from `www.mitov.com`.
I am planning to write more articles on **DTI,** and maybe even an article
on how you can define new **Visuino** components just using text editor,
thanks to the new **DTI.**
The **Blaise Pascal Magazine** web site allows you to write comments.
If you are interested in any of these topics,
or if you want to learn more about **Video Processing,**
Audio Processing, Signal Processing, Computer
Vision, AI, SCADA Industrial Control systems,
programming **Arduino,**
or **Raspberry Pi** with **Visuino**
or maybe programming autonomous
or remote controlled by **Delphi** robots,
please leave comments
with your suggestions,
and I will try to write articles
on the suggested topics.
Hope to see
your comments soon...

The Chameleon you see here
is a colouring-game.
If you go to the website
`https://www.blaisepascalmagazine.eu/colorgame/`
you'll find a menu that allows you to change
the colour of the Chameleon in the colour of your choice.
You can of course revert to the original colour.

There are special areas that will pop up a message,
from `Mitov.com`

Advertisement

Blaise Pascal Magazine 94/95 2021

By David Dirkse

starter .... expert

Flippos are small cardboard or plastic discs holding the printed image of a football player, movie star or comic hero.
In the '90s potato chips producer Smith added a flippo to their packets which made flippo collection a craze.
This is the flippos collector problem:
After collecting a certain amount of flippos, what is the probability to possess a complete series?

This article consists of three parts:
❶ Basic theory behind :
   an intro to combinatorics
❷ The solutions of the flippo problem
❸ Description of the program

Theory (a short course in combinatorics)
Combinatorics is the part of math where possibilities are counted.

Examples are:
- the number of boards to choose from a group of people (chairman, secretary, cashier, members)
- the ways to connect railway cars (1st, 2nd class, luggage-, restaurant carriage)
- the number of sequences in which to visit 20 customers
- the number of ways to travel from A to B

Combinatorics is the base of probability as this is the division of wanted- and total outcomes of a process. In the selection of objects or the counting of events the question is
❶ May we select an object from a set more than once?
❷ Is the sequence of selection important?

Answering yes or no results in four cases:

CASE 1: MULTIPLE SELECTION,
SEQUENCE IS IMPORTANT.
This is the case with number systems.
In the decimal system using 6 digits we can make numbers from `000000` to `999999`
which are $10^6 = 1000000$ rows of 6 digits.
In general, having `N` different digits, there are N^K different numbers of `K` digits.

Example:
A 32-bit word in computer memory may hold
$2^{32} = 4294967296$ numbers,
from 0 to $2^{32} - 1 = 4294967295$

Example:
Using characters A,B,D,E,F
we can make 5^8 words of 8 characters,
from `AAAAAAAA` to `FFFFFFFF`.

CASE 2: SINGLE SELECTION, SEQUENCE IS IMPORTANT
This is the case with permutations.
 A permutation simply is a sequence of objects or elements. When visiting 20 customers there are $20x19x18x...x3x2x1 = 2*4 \times 10^{18}$
sequences. After each visit the choice for the next customer is one less.
`N! = (N).(N-1)(N-2)......(3)(2)(1)`
say `N` factorial for `N!`
which is the number of sequences in which we can arrange `N` objects.
The characters `A,B,C,D` can be arranged to make `4! = 4*3*2*1 = 24` sequences
from `ABCD` to `DCBA`.

N**ow here follows a very important rule:**
How many sequences are possible if some elements are the same?
Let the elements be `AAAABCDE`.
The trick is making the `A's` different at first by applying an **index:** (like in an array)
$A_1A_2A_3A_4$`BCDE` are 8 elements which can make `8!` sequences.
$A_1A_2A_3A_4$ are 4 elements which can make `4!` sequences however, because they are the same only one sequence `AAAA` (`4!` times less) is possible.
So, `8!` must be divided by `4!` the answer is
`8! / 4! = 8*7*6*5 = 1680` sequences.

Example:
From `15` people a committee is chosen (chairman, secretary, cashier).
Call the chairman A, secretary B, cashier C and call not chosen people `N`.
Then the number of boards possible is the way we can make sequences of characters
`ABCNNNNNNNNNNNNN`.
Applying the rule before:
`15!/12! = 15*14*13 = 2730`.

Example:
In how many ways can we arrange a train if there are `8` 2nd class, `3` 1st class, 3 luggage and 2 restaurant carriages?
Call first class `F`, second class `S`, luggage `L` and restaurant carriage `R`.

How many sequences are possible of the characters `FFFSSSSSSSSSLLLRR` ?
`16!` must be divided by `8!` (the second class) also by `3!` (first class) `3!` (luggage) and `2!`(restaurants)
`16!/(8!.3!.3!.2!) =`
`16*15*14*13*12*11*10*9/(6*6*2) =`
`7207200` possibilities.

CASE 3 : SINGLE SELECTION, SEQUENCE UNIMPORTANT
This kind of choice is called a **combination.**

Example:
After a party a team of `4` people (from a group of 12) must be selected to clean the area.
We apply the character `Y` to people selected and a `N` to not selected people.
The number of choices (combinations) is the possible number of sequences of characters `YYYYNNNNNNNN`
which is `12!` divided by `4!` and also by `8!`.
`12!/(4!.8!) =`
`12.11.10.9/(4.3.2.1) = 495.`
Combinations are very common in mathematics so a special notation exists:

Combinations

choice of K elements from N =

$$\binom{N}{k} = \frac{N!}{(N-K)!.K!}$$

examples:

$$\binom{5}{3} = \frac{5!}{3!.2!} = \frac{5.4.3.2.1}{3.2.1.2.1} = 10$$

$$\binom{10}{0} = 1 \qquad \binom{20}{1} = 20$$

Example:
A computer code exists of 3 zero bits and 4 one bits.
How many codes are possible?
Just the amount of codes as there are sequences of `0001111` which are the number of combinations of `4` out of
`7 = 7!/(3!4!) = 35`.

Example: (Newton binomium)
$(a+b)^2 = aa + ab + ba + bb = a^2 + 2ab + b^2$
$(a+b)^3 = aaa + aab + aba + abb + baa + bab + bba + bbb = a^3 + 3a^2b + 3ab^2 + b^3$

We notice that `a`,`b` act as digits of a counter.

So, in $(a+b)^{10}$ the term a^7b^3 originates from `aaaaaaabbb` and the number of occurrences are the combinations of 7 out of 10.
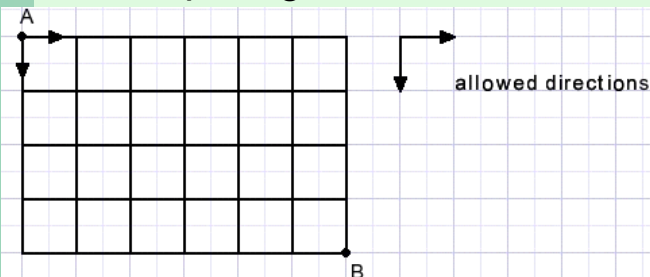The term has coefficient `10!/(7!*3!) = 120`

Newton binomium

$$(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + ... + \binom{n}{n}b^n$$

$$= \sum_{i=0}^{n} \binom{n}{i}a^{n-i}b^i$$

Example:
below is shown a roadmap.
Question is how many roads exist from left top A to right bottom B.
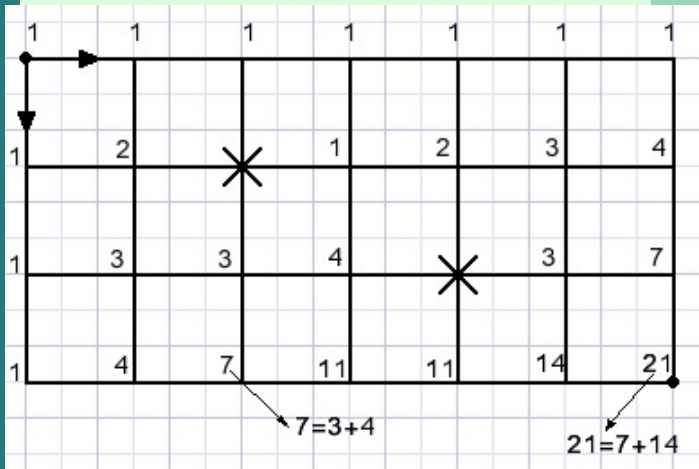

allowed directions

roads A to B

$$\binom{10}{6} = 210$$

Call a horizontal direction `H` , vertical direction V, a possible road is `HHHHHHVVVV`
but all other combinations are fine.

How to proceed if some roads are blocked?
Write a 1 at the starting position, we arrived here in one way.
For each road crossing, write the sum of the numbers of the preceding crossings.

7=3+4

21=7+14

CASE 4: MULTIPLE SELECTION, SEQUENCE UNIMPORTANT

This is the case where digits may be (re) selected but the position is unimportant.
In this case the number `54481` is equal to `14458`.

I call this the zoo problem.
Say we want to build a zoo with 12 animals, the choice is of apes `(A)` bears `(B)` crocodiles `(C)` and eagles `(E)` .

How many zoos are possible? A possible zoo could be `AAA | BBBB | CCC | EE`
Call | a fence to separate the animals.
Then the number of zoos is the number of ways we can place the three fences:
`AAAAAAA||CCCCC|` would be a zoo with 7 apes, no bears, 5 crocodiles and no eagles.

Because the location of the animals is fixed, we no longer need the characters `A,B,C,D,E` and we write `X`.
The first zoo becomes:
`XXX | XXXX | XXX | XX`.
`Animals + fences + 12 + 3 = 15`.

Three fences have to be placed where the choice is of 15 places.
Note that the number of fences is the number of animal types – 1.

The number of zoos are the combinations of `3 out of 15` so `15!/(12!3!) = 455`.

Note: the selection is from places `XXXXXXXXXXXX---` where `---`are the fences.

In general:
The number of `K` choices from `N` re-selectable objects where the sequence is unimportant is:

$$\binom{N+K-1}{N-1}$$

Example:
In how many ways can we paint 15 eggs if the choice is of 7 colors?
Comparing with the zoo problem:
Eggs = animal count, colors are animal types.
Align the eggs and place cardboard between the colors.
`K = 15` (eggs). `N = 7` (colors).

$$\binom{N-1+K}{N-1} = \binom{7-1+15}{7-1} = \binom{21}{6} = \left(\frac{21!}{15!6!}\right) = 54264$$

So far for the theory.

THE FLIPPOS COLLECTOR PROBLEM

Three calculation methods are presented:
❶ Straight counting.
❷ A formula.
❸ A simplified formula with limited accuracy.

Straight counting
The flippos in a series are called `1,2,3,...,N` where `N` is the number of different flippos of a series.

This method was developed first to serve as validation for later developed formulas.
We simply write all possible selections of `K` flippos and count the number of selections that contain a complete series.

Say `N=3` and `K=5`,
then some of the `35 = 234` rows are:
```
1 1 1 1 1
1 1 1 1 2
. . .
1 3 3 3 3
2 1 1 1 1
. . .
3 3 3 3 3
```
A "good" row is `3 3 1 1 2`
A "wrong" row is : `2 2 1 1 2`

For `K` flippos from a collection of `N`, `NK` rows of `K` flippos are generated.
If `G` "good" rows are counted then the probability to posses a complete series in `K` flippos is:

$$\frac{G}{N^K}$$

This method is very time consuming so the limit is set to small numbers
`{NK < 250.000.000}`

A formula
This provides for a fast and accurate calculation. Say `A(1)` is the number of rows where flippo 1 is absent.
Such a row was generated by choosing `K` times from (N-1) flippos, so `(N-1)K` rows result.
`A(1+2)` is the number of rows with missing flippos 1 OR 2 and `A(1*2)` the number of rows with missing flippos 1 `AND` 2.
Read "+" as OR , "*" as AND.
Then `A(1+2) = A(1) + A(2) - A(1*2)` the last term corrects for double counting.

A(1.2)
A(1) A(2)
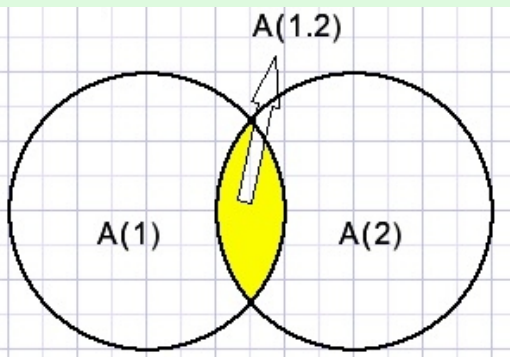
Applying this rule further:
```
A(1+2+3) =
A((1+2)+3) =
A(1+2) + A(3) - A((1+2)*3)
A((1+2)*3) = A(1*3+2*3) = A(1*3) +
A(2*3) - A(1*2*3)
```

These rules may be visualized by drawing overlapping areas as we noticed before.

Summing up :
```
A(1+2+3) = A(1) + A(2) + A(3) - [A(1*2
+ A(1*3) + A(2*3)] + A(1*2*3)
```

```
A(1) = A(2) = ...= A(K) are N terms.
A(1*2) =
A(1*3) = A(2*3) = .... = A((k-1),K)
```
are just as much terms as we can select 2 flippos out of `N`.
This results in a formula for the amount of incomplete rows:

$$A(1+2+..+N) = \binom{N}{1}(N-1)^K - \binom{N}{2}(N-2)^K + \binom{N}{3}(N-3)^K - ....$$

$$= \sum_{i=1}^{N} (-1)^{i-1} \binom{N}{i} (N-i)^K$$

The number of "good" rows is
`N^K - A(1+2+...+N)`
Division by `NK` results in the probability for a

A shorter formula
This formula provides an approximation which is only accurate for large numbers of `K >> N`
We choose a row of `K` flippos without flippo 1.
The probability for this row is:

$$\left[\frac{N-1}{N}\right]^K$$

So, the probability for a row with at least 1 flippo of type -1- is:

$$1 - \left[\frac{N-1}{N}\right]^K$$

Repeating this formula for flippos type `2..N` the probability of a complete series is:

$$\left\{1 - \left[\frac{N-1}{N}\right]^K\right\}^N$$

Above, we multiplied chances that were not completely independent which results in an error. For large values of K however this error is small. The formula may be simplified:

$$e = \lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^{x} = \lim_{x \to 0}(1+x)^{\frac{1}{x}}$$

$$e^{-1} = \lim_{x \to 0}(1 - (-x))^{\frac{1}{-x}}$$

$$\lambda = \left(\frac{N-1}{N}\right)^{K}$$

$$\left[1 - \left(\frac{N-1}{N}\right)^{K}\right]^{N} = (1 - \lambda)^{N} = \left[(1 - \lambda)^{\frac{1}{\lambda}}\right]^{\lambda N} \approx e^{-\lambda N}$$

Table below shows some values for the case of N = 5
The "formula" column is accurate,
the "e-power" column is calculated using the approximation e power formula.

K	Formula	e-power approximation
5	3.8%	19.4%
10	52.5%	58.5%
20	94.3%	94.4%

THE PROGRAM

This Delphi project has only one unit. UpDown components allow for the selection of variables N and K.
Counting , Formula or short formula approximation may be selected by Speedbutton components.
The speedbuttons use their tag (set to 1,2,3) to identify themselves: they share the OnClick event.
The counting method is not selectable for large numbers, the calculations simply would take too long.
Variables are of type double, which allows for 15 digit long numbers. The counting method uses an array (AK) of bytes to represent a selection of K elements.

Flippo series investigator

complete collection (N) 5
collected (K) 10
go
stop

───method───
counting | formula | K >> N

incomplete series
4662625

probability complete series
52,255 %

$$1 - \frac{4662625}{9765625} =$$

total series

finished

```pascal
const maxN = 10; //maximal number of different flippo types
    maxK = 50; //maximal number of collected flippos
var AK : array[1..maxK] of byte;
  K,N : byte;
  stopflag : boolean;
  calcmethod : byte;//1: count 2:formula  3:large K
```

Initially AK[ ] is set to 1111111...1111111

```pascal
function incAK : boolean; //increment AK array
  var i : byte;
  carry : boolean;
begin
  i := 1;
  repeat
   inc(AK[i]);
   carry := AK[i] = N + 1;
   if carry then
    begin
     AK[i] := 1;
     inc(i);
    end;
  until (carry = false) or (i = K+1);
  result := i <= K;
end;
```

True is returned if the counter did not overflow.
Before incrementing, the counter must be checked for a complete series.

```pascal
function checkAK : boolean;
//return true if complete series in AK
var AN : array[1..maxN] of boolean;
  d,i : byte;
begin
  for i := 1 to N do AN[i] := false;
  for i := 1 to K do
   begin
    d := AK[i];
    AN[d] := true;
   end;
  result := true;
  i := 1;
  while result and (i <= N) do
   begin
    result := result and AN[i];
    inc(i);
   end;
end;
```

The code above is called by procedure count to do the job.
Please see the source code for details.

The "formula" method needs the calculation of faculties, powers and of course the number of combinations.

```pascal
function power(base : double; x : byte) : double;
var i : byte;
begin
  result := 1;
  for i := 1 to x do result := result * base;
end;
function faculty(NN : byte) : double;
var i : byte;
begin
  result := 1;
  for i := 1 to NN do result := result * i;
end;
function combinations(NN,KK : byte) : double;
//calculate NN over KK combinations
var i : byte;
```

Procedure formula calls above code to do the job. Finally, the "**largeK**" procedure calculates the e-power formula:

```pascal
procedure largeK;
//calculate approximation for K >> N
var p,lambda : double;
begin
  lambda := power((N-1)/N,K);
  p := exp(-lambda*N);
  form1.Ppercentage.Caption := formatfloat('##0.##',100*p);
end;
```
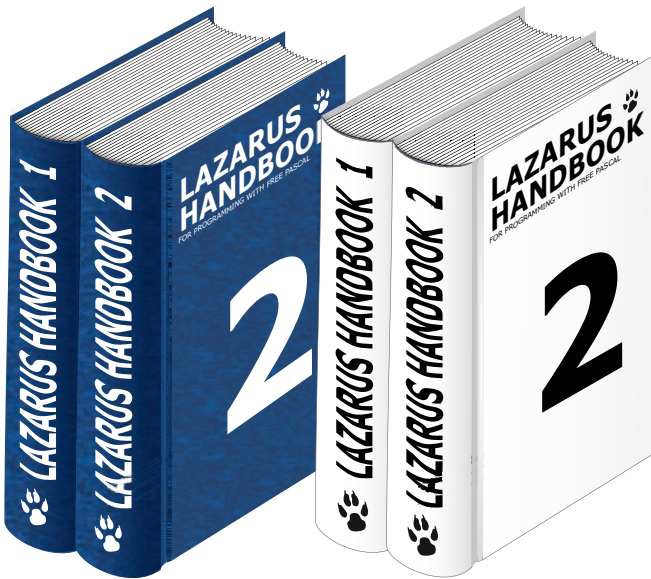
NOTES:
The "counting" method has a possibility to interrupt the counting. A click on the **STOP BitBtn** sets the STOP flag which terminates counting.
This concludes the description of the flippo collectors problem.
Please refer to the source code for details.

Implementing APIs for Chromium
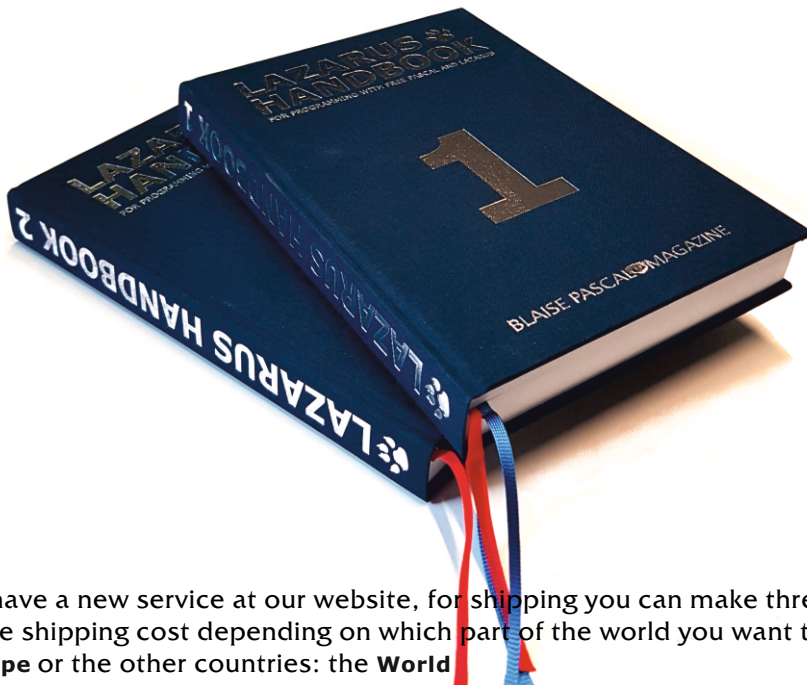
By Michaël Van Canneyt

starter ────────────── expert

ABSTRACT
In the previous article about embedding Chromium, we showed how to embed Chromium in your application. In this article, we go a step further, and we show how to add APIs to the Chromium environment, and how these APIs can be used in your Javascript or Pas2JS code.

❶ INTRODUCTION

In the previous article, we showed how you can embed Chromium in your Lazarus- (or Delphi, the procedure is the same) application.

We also showed how you can load files using a private protocol. This can be used to show any website, or to limit the shown HTML files shown to the files you choose.

However, the same can be done (can more or less) by starting the browser e.g. in **kiosk mode*** as a separate process: the possibilities are limited to what the browser offers you.

It becomes really interesting when you start adding possibilities to the embedded browser, that allows to interact with the user environment. It is possible to add file management, **tray icon management**, menu management, or virtually anything you want to add to the Browser environment:
You can create Javascript objects with methods, and make these available to the HTML and Javascript running in the browser windows.

In this article, we'll show how to do this:
we'll demonstrate how to add separate logging to the browser, or how to let the browser interact with a tray icon and popup menu that lives in your application.

Obviously, much more could be done, but the examples serve just to show how to do these things.

Kiosk software is the system and user interface software designed for an interactive kiosk or Internet kiosk enclosing the system in a way that prevents user interaction and activities on the device outside the scope of execution of the software. This way, the system replaces the look and feel of the system it runs over, allowing for customization and limited offering of ad-hoc services. Kiosk software locks down the application in order to protect the kiosk from users which is specially relevant under, but not only limited to, scenarios where the device is publicly accessed such libraries, vending machines or public transport.

Kiosk software may offer remote monitoring to manage multiple kiosks from another location. An Email or text alerts may be automatically sent from the kiosk for daily activity reports or generated in response to problems detected by the software.
Other features allow for remote updates of the kiosk's content and the ability to upload data such as kiosk usage statistics. Kiosk software is used to manage a touchscreen, allowing users to touch the monitor screen to make selections. A virtual keyboard eliminates the need for a computer keyboard.

❷ ARCHITECTURE

Adding Javascript classes to the Chromium browser is not so difficult. CEF (Chromium Embedded Framework) has a rich API (Application Programming Interface) for this, and the complete API is available to you through **CEF4Delphi.**

Basically, you build a Javascript object using the appropriate APIs, attach some methods or properties to it with some **callback** methods, and that is it.
(*A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.*)

However, it becomes more complicated when you need to handle a GUI in your Javascript objects.
For instance, you could decide to show a **file-open** dialog, or allow access to a menu, or anything else that requires access to the GUI (Graphical User Interface).
The reason it becomes more complicated is the browser is running in a separate process: when you start the CEF browser, it creates a new process (called the RENDER process), and everything connected with the browser is running in that process. This includes the Javascript and any classes you make available to the browser.

The code for your Javascript class will be running in the RENDER process, but your GUI components will live in your original program: called the BROWSER process (somewhat of a misnomer).

So, if your Javascript classes wish to change the GUI (show or hide a tray icon), or if an event in the GUI (a menu click) must be communicated to the Javascript, this will involve communication between the RENDER process and the BROWSER process.

This communication happens with messages, and is **asynchronous.**

Asynchronous Messaging is a communication method where participants on both sides of the conversation have the freedom to start, pause, and resume conversational messaging on their own terms, eliminating the need to wait for a direct live connection.

This is schematically depicted in Figure 1 on page 2 of this article (Below).
Imagine you wish to make a special log call available to the Javascript code, which sends the log message to a memo on the main window.
This involves creating a class which will live in the browser-render-process.
When the Javascript log method is executed, the method in the class will receive the logged items.
To actually display the message, it needs to send this to the GUI process using a message (depicted by the blue, upward arrow).
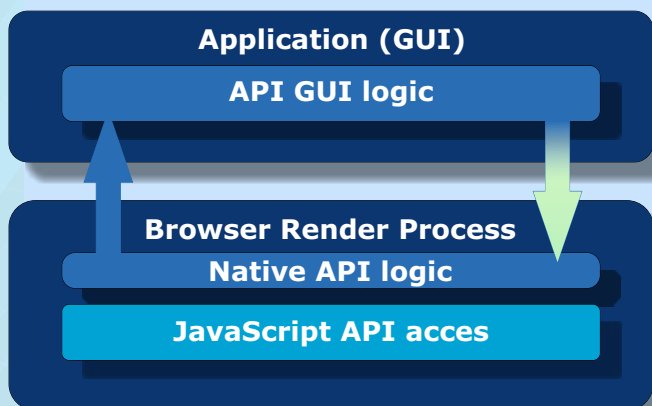

Figure 1: Interprocess communication for CEF

The reverse is also true: If there is an API that allows the browser Javascript code to respond to a menu click (for example a popup menu item of a tray icon), and the user clicks a message, there must be some way that the GUI process communicates this to the browser RENDER process (depicted by the green, downward arrow), and then the classes that implement the Javascript API will communicate this to the browser.

All this requires quite some code, but luckily the CEF API has a mechanism for sending messages between the two processes. In the below, we show how to do this.

We'll build on one of the previous examples: we create a small application with 2 APIs: one simply replaces console.log with a custom method: it will display the logged data in a memo. Since we implement a logging method, we'll also log unhandled Javascript exceptions, as a useful debugging tool.

The second API is an object that represents a Tray icon, managed by the program. The API is an object, with a property that controls the visibility of the tray icon – not surprisingly, the property is called 'visible'. It also has 2 methods: one to add a menu item to the popup menu of the tray icon, the second to remove the menu item.

In JavaScript, these 2 APIs would look like this:

```
window.trayIcon = {
    visible : boolean,
    addMenuItem : function(aCaption, ACallBack) {
    },
    removeMenuItem : function (aID) {
    }
};
window.appLog = function() {

};
```

So, how to implement these in CEF? The CEF offers a rich API for representing Javascript values. The basis of this is the `ICefv8Value` interface, defined in the **uCEFInterfaces** unit, like all other interfaces offered by CEF. It allows to check the value type, get or set the value, or, if the interface represents an object, query the members of the object.

❸ CREATING A JAVASCRIPT FUNCTION

Functions in Javascript are values like any other, so if we want to create a logging function, it stands to reason that this will be a `Function` value.
In CEF4Delphi, the `TCefv8ValueRef` class implements the `ICefv8Value` interface.
So our logging function `appLog` starts with a `TCefv8ValueRef` instance:
We create it like this, in the `CreateLoggerObject` function:

```
Function CreateLoggerObject : ICefv8Value;
begin
  Result:=TCefv8ValueRef.NewFunction(SFuncDoLog, TLogHandler.Create);
end;
```

The `NewFunction` creates a function object.
The first parameter is the name, the second parameter is the function handler interface:
This must be a `ICefv8Handler` interface.
This interface will be called when the function is invoked in Javascript.
In our logging function, the `TLogHandler` class is created. This is a descendent of the `TCefv8HandlerOwn` class in CEF4Delphi, which implements `ICefv8Handler` for us.
This class has only 1 method that we must override, aptly named `Execute`:

```
TLogHandler = class(TCefv8HandlerOwn)
function Execute(const name: ustring;
              const obj: ICefv8Value;
              const arguments: TCefv8ValueArray;
              var retval: ICefv8Value;
              var exception: ustring): Boolean; override;
end;
```

The meaning of the parameters should be clear from their names.

In the `Execute` function, the logic of the function must be implemented.

The Javascript `console.log` function accepts any number of arguments, and writes a string representation of these arguments to the console. Here we mimic this behaviour. We start by creating a string representation of the arguments, separated by spaces.
As explained above, the code of `TLogHandler` is executed in the RENDER process. To actually display the string, we must send it to the BROWSER process.

This can be done with the `ICefProcessMessage` interface, which is implemented by the `TCefProcessMessageRef` class. So we must create an instance of this class,
and attach the string to it: an arbitrary amount of data can be attached to a message, the `ArgumentList` property of the message must be used for this.
Every message is named, and the BROWSER process can handle the messages by checking the name. In our program, the names of the messages are defined as pascal constants. For the log message, the name is in the `SMsgDisplayLogMessage` constant:

```
function TLogHandler.Execute(const name: ustring;
  const obj: ICefv8Value;
  const arguments: TCefv8ValueArray;
  var retval: ICefv8Value;
  var exception: ustring): Boolean;

Var
  msg: ICefProcessMessage;
  i,j : Integer;
  S : uString;
begin
  Result:=True;
  msg := TCefProcessMessageRef.New(SMsgDisplayLogMessage);
  msg.ArgumentList.SetSize(1);
  S:='';
  For I:=0 to Length(Arguments)-1 do
    begin
      if I>0 then S:=S+'';
      S:=S+CefValToString(Arguments[i]);
    end;
  msg.ArgumentList.SetString(0,S);
  SendProcessMessage(PID_BROWSER,msg);
end;
```

The last 2 statements attach the constructed string to the message, and send it to the BROWSER process using the `SendProcessMessage` method of the `iCEFFrame` interface:
this interface represents a HTML frame of a webpage. The `SendProcessMessage` is a small helper method that encapsulates

`TCefv8ContextRef.Current.Browser.MainFrame.SendProcessMessage`

This long expression is a rather complex way of getting a reference to the main frame of the HTML page from which the Javascript function was called.
How the BROWSER process must respond to this message, we will see later in this article.
The `CefValToString` function transforms a `iCEFv8Value` to a string, it uses the methods of `iCEFv8Value` to inspect the value type and convert it in an appropriate way to a string:

```
Function CefValToString(A : ICefv8Value) : String;
var i,j : Integer; L : TStrings;
begin
  if A.IsString then Result:=A.GetStringValue
  else if A.IsBool then
    Result:=BoolToStr(A.GetBoolValue)
  else if A.IsInt then
    Result:=IntToStr(A.GetIntValue)
  else if A.IsDouble then
    Result:=FloatToStr(A.GetDoubleValue)
  else if A.IsNull then
    Result:='null'
  else if A.IsArray then begin
    Result:='[';
    for J:=0 to A.GetArrayLength-1 do
      begin
        if J>0 then Result:=Result+', ';
        Result:=Result+CefValToString(A.GetValueByIndex(J));
      end;
  Result:=Result+']';
  end
  else if A.IsObject then begin
    Result:='{';
    L:=TstringList.Create;
    try
    A.GetKeys(L);
    For J:=0 to L.Count-1 do begin
    if J>0 then Result:=Result+', ';
    Result:=Result+'"'+L[i]+'": '+CefValToString(A.GetValueByKey(L[i]));
    end;
  finally l.Free;
  end;
  Result:=Result+']';
  end
else
Result:=Result+'[Cannot display this value]';
end;
```

Note that this function recursively calls itself in the case of Javascript arrays and objects.

❹ CREATING A JAVASCRIPT OBJECT
Creating a Javascript object is not much different from creating a function: we define it as a value of type object (using the `NewObject` class method), and attach the definition of the `visible` property and the methods to add or remove menu items to it.
To control access to the properties of an object, the `ICefV8Accessor` interface must be used.
This interface must be passed to the `NewObject` method. We'll create a class `TTrayIconAccessor` to handle the access for our trayIcon object:
The CEF4Delphi framework defines a `TCefV8AccessorOwn` class which has the necessary methods for the `ICefV8Accessor` interface, and if we make a descendent of this class, we must simply override these methods.

Adding a property to a Javascript object can be done using the `SetValueByKey` method of `iCEFV8value`.
This method takes 3 parameters:
the name, the initial value (a boolean in our case) and a set of attributes (some OR-ed integer values).
Object methods are functions, and functions are Javascript values like any other.
So, in order to define a method for an object, we must simply add a property with the name of the method and supply a function value as initial value.
As seen above, functions can be implemented with an instance of a handler class.
Because the `Execute` method of the handler class receives the name of the function that is called, we can use a single handler class to handle both the `addMenuItem` and `removeMenuItem` methods:
The `TTrayIconHandler` class.
Armed with the 2 classes `TTrayIconHandler`

Implementing APIs for Chromium

and **TTrayIconAccessor**, we can now
construct our **trayIcon** object in the
CreateTrayIconObject class. The main
method to define a property is
SetValueByKey:

```pascal
Function CreateTrayIconObject : ICefv8Value;
Var
   TempAccessor : ICefV8Accessor;
   TempHandler : ICefv8Handler;
   Ref : ICefv8Value;
begin
   TempAccessor := TTrayIconAccessor.Create;
   TempHandler:= TTrayIconHandler.Create;

   // Create object
   Result:=TCefv8ValueRef.NewObject(TempAccessor,nil);

   // Add visible property
   if not Result.SetValueByKey(SPropVisible,
                               TCefv8ValueRef.NewBool(False),
                               V8_PROPERTY_ATTRIBUTE_NONE) then
       Raise Exception.Create('Could not set visible attribute');

   Result.SetValueByAccessor(SPropVisible,
                               V8_ACCESS_CONTROL_DEFAULT,
                               V8_PROPERTY_ATTRIBUTE_NONE);

   // Add addMenuItem method
   Ref:=TCefv8ValueRef.NewFunction(SFuncAddMenuItem,TempHandler);
   if not Result.SetValueByKey(SFuncAddMenuItem,
                               Ref,
                               V8_PROPERTY_ATTRIBUTE_NONE) then
       Raise Exception.Create('Could not add addMenuItem function');

   // Add removeMenuItem method
   Ref:=TCefv8ValueRef.NewFunction(SFuncRemoveMenuItem,TempHandler);
   if not Result.SetValueByKey(SFuncRemoveMenuItem,
                               Ref,
                               V8_PROPERTY_ATTRIBUTE_NONE) then
     Raise Exception.Create('Could not add addMenuItem function');
end;
```

Object properties have a lot of attributes in
CEF, but we don't need those, so we pass
V8_PROPERTY_ATTRIBUTE_NONE for all of
them.
Access to the **visiblity** property is handled
by the following class:

```pascal
TTrayIconAccessor = class(TCefV8AccessorOwn)
private
  FVisible : Boolean;
  procedure SendBrowserShowHideTrayIcon(aValue: Boolean);
protected
  function Get(const name: ustring; const obj: ICefv8Value;
              var retval : ICefv8Value;
              var exception: ustring): Boolean; override;
  function Set_(const name: ustring; const obj,
              value: ICefv8Value;
              var exception: ustring): Boolean; override;
end;
```

It is quite a simple class: it has methods `Get`
and `Set_`, which are called whenever a
property is read or written, much like the
getter and setter of properties in Pascal.
The name of the property is passed to these
methods. In our case, only the 'visible'
property is handled.
The following is then pretty straightforward:

```pascal
function TTrayIconAccessor.Get(const name : ustring;
         const obj : ICefv8Value;
         var retval : ICefv8Value;
         var exception : ustring): Boolean;
begin
  Result:=(name=SPropVisible);
  if Result then
     retval:=TCefv8ValueRef.NewBool(FVisible);
end;
```

The setter is slightly more complicated:
we must not only check the name, but also the
value type:

```pascal
function TTrayIconAccessor.Set_(const name : ustring;
                               const obj : ICefv8Value;
                               const value : ICefv8Value;
                               var exception : ustring): Boolean;
begin
    Result:=(name=SPropVisible);
    if Result then
    begin
    if value.IsBool then
      begin
      FVisible:=value.GetBoolValue;
      SendBrowserShowHideTrayIcon(FVisible);
      end
    else
    exception := 'Invalid value type, expect boolean';
    end
end;
```

Note that we set the `exception` parameter if
something is wrong with the passed value:
This will cause a Javascript exception to be
raised.
To actually show (or hide) the tray icon, the
BROWSER process must be notified. For this,
we send again a message, and this is done in the
`SendBrowserShowHideTrayIcon` method.
The process is similar to the way the browser is
notified that a log message is sent.

```pascal
procedure TTrayIconAccessor.SendBrowserShowHideTrayIcon(aValue : Boolean);
var
  msg: ICefProcessMessage;
begin
  msg := TCefProcessMessageRef.New(SMsgSetTrayIconVisibility);
  msg.ArgumentList.SetBool(0,aValue);
  SendProcessMessage(PID_BROWSER, msg);
end;
```

The handler class for the tray icon much resembles the function handler for the log function:

```
TTrayIconHandler = class(TCefv8HandlerOwn)
Private
  Class Var FMenu : TTrayMenu;
  function AddMenuItem(aCaption: uString; AHandler: ICefv8Value;
  AContext : ICefV8Context): Integer;
  function RemoveMenuItem(aID: Integer;
  AContext: ICefV8Context): Integer;
  class procedure CheckMenu; static;
protected
  class procedure CallTrayMenuCallBack(aID : integer); static;
  function Execute(const name: ustring; const obj: ICefv8Value;
                                      const arguments: TCefv8ValueArray;
                                      var retval: ICefv8Value;
                                      var exception: ustring): Boolean; override;
end;
```

Again, the Execute method is the entry point when the Javascript environment needs to call the methods: It checks the name of the called function:

```
function TTrayIconHandler.Execute(const name: ustring;
  const obj: ICefv8Value;
  const arguments: TCefv8ValueArray;
  var retval: ICefv8Value;
  var exception: ustring): Boolean;

Var
  aLen, aID : Integer;
begin
  Result:=True;
  aLen:=length(arguments);
Case name of
  SFuncAddMenuItem:
    begin
      if (aLen>1)
      and arguments[0].IsString
      and arguments[1].IsFunction then
        begin
        aID:=AddMenuItem(arguments[0].GetStringValue,
                          arguments[1],
                          TCefv8ContextRef.Current);
        retVal:=TCefv8ValueRef.NewInt(aID);
        end
      else
      exception:=
        ,Need 2 arguments: caption and callback';
      end;
  SFuncRemoveMenuItem:
    begin
      if (aLen>0) and arguments[0].IsInt then
      begin
        aID:=RemoveMenuItem(arguments[0].GetIntValue,
                          TCefv8ContextRef.Current);
        retVal:=TCefv8ValueRef.NewInt(aID);
        end
      else
        exception:='Need 1 arguments: menu item ID';
      end;
    else
        Result:=False;
end;
```

The **AddMenuItem** and **RemoveMenuItem** functions do the actual work: they update the local copy of the tray icon menu, and then send a message to the BROWSER process. The local copy of the menu is a simple global collection with ID, caption, handler and context. These fields are needed to keep the various event handlers that have been registered in the Javascript: this is the **aHandler** argument.

The **aContext** parameter (and property) is necessary to be able to call the correct handler when the BROWSER process sends a message notifying that the menu item is clicked:

```
function TTrayIconHandler.AddMenuItem(aCaption : uString;
AHandler : ICefv8Value; AContext : ICefV8Context): Integer;
Var
  M : TTrayMenuItem;
  msg: ICefProcessMessage;
begin
  CheckMenu;
  M:=FMenu.AddMenu(aCaption);
  M.Handler:=aHandler;
  M.Context:=aContext;
  Result:=M.ID;
  // Send message to browser process.
  msg := TCefProcessMessageRef.New(SMsgAddTrayMenuItem);
  msg.ArgumentList.SetSize(2);
  msg.ArgumentList.SetInt(0,M.ID);
  msg.ArgumentList.SetString(1,M.Caption);
  SendProcessMessage(PID_BROWSER, msg);
end;
```

As you can see, the last statement again sends the message to the BROWSER process. The **RemoveMenuItem** is similar, but it needs less arguments:

```pascal
function TTrayIconHandler.RemoveMenuItem(aID: Integer;
AContext: ICefV8Context): Integer;
Var
msg: ICefProcessMessage;
begin
CheckMenu;
FMenu.RemoveMenu(aID);
Result:=0;
// Send message to browser process.
msg := TCefProcessMessageRef.New(SMsgRemoveTrayMenuItem);
msg.ArgumentList.SetSize(1);
msg.ArgumentList.SetInt(0,aID);
SendProcessMessage(PID_BROWSER,msg);
end;
```

With this, we have implemented a Javascript object that can be manipulated from within a HTML Page displayed in the embedded browser.

❺ ADDING THE IDENTIFIERS TO THE BROWSER

The previous paragraphs showed how to construct Javascript functions and objects using the classes that CEF4Delphi makes available, but it did not yet make the browser aware of these classes. This must be handled explicitly. Also, we still need a mechanism to react on the actual `OnClick` of the tray icon's popup menu: from the architecture sketched above, it should be clear that the BROWSER process will have to notify the RENDER process with a message when a click happens.
We also want to do 2 extra things:
- Replace `console.log` with our own handler, so all log messages are sent to the Console.
- Log a message whenever a Javascript exception occurs. This is useful for debugging.

The place to set up the necessary callbacks for all this is in the `CreateGlobalCEFApp` routine we developed in the previous article. In this routine, we set some additional event handlers:

```pascal
procedure CreateGlobalCEFApp;
begin
  if GlobalCEFApp <> nil then exit;
  GlobalCEFWorkScheduler := TCEFWorkScheduler.Create(nil);
  GlobalCEFApp:= TCefApplication.Create;
  GlobalCEFApp.ExternalMessagePump:= True;
  GlobalCEFApp.MultiThreadedMessageLoop:= False;
  GlobalCEFApp.OnScheduleMessagePumpWork:=
            @GlobalCEFApp_OnScheduleMessagePumpWork;
  GlobalCEFApp.OnRegCustomSchemes:=@CEFRegisterCustomSchemes;
  // New handlers
  GlobalCEFApp.OnContextCreated:=@HandleNewContext;
  GlobalCEFApp.OnWebKitInitialized:=@HandleWebkitInitialized;
  GlobalCEFApp.OnUncaughtException:=@HandleJSException;
  GlobalCEFApp.OnProcessMessageReceived := @HandleProcessMessage;
  // End of new handlers
  {$IFDEF LINUX}
  GlobalCEFApp.DisableZygote := True;
  {$ENDIF}
end;
```

From the comments, we can see that 4 extra events have been assigned. We'll discuss them one by one.
The `OnContextCreated` event is fired when a new browser context is created. This is where you can introduce new Javascript identifiers for that particular browser.
We use the 2 functions we created in the above to create an instance of the objects we wish to expose. Any global Javascript object or identifier is actually attached to the `Window` object. So, we must attach our new identifiers to the global `Window` object. Since this is a Javascript object like any other, the code to do so is no different from the code to add methods and properties to an object, except that in this case the object is the `Window` object, which is made available to us through the `context.Global` value:

```pascal
procedure HandleNewContext(const browser: ICefBrowser;
const frame: ICefFrame;
const context: ICefv8Context);
Var
logger,
trayIcon: ICefv8Value;
begin
trayIcon:=CreateTrayIconObject;
logger:=CreateLoggerObject;
With Context.Global do
begin
if not SetValueByKey(STrayIcon,trayIcon,
V8_PROPERTY_ATTRIBUTE_NONE) then
Raise Exception.Create(
            'Could not set global tray icon object');
if not SetValueByKey(SFuncDoLog,
logger,
V8_PROPERTY_ATTRIBUTE_NONE) then
Raise Exception.Create(
            'Could not set global log function');
end;
end;
```

That's all there is to it.
CEF offers the possibility to execute some custom Javascript when the browser window is created and ready, but before all other Javascript is executed. This is called an 'extension'.
Extensions can be registered using the `CefRegisterExtension` method of CEF. We will use this to reroute the `console.log` function to our own appLog function. The place to register an extension is in the `OnWebKitInitialized` event:

```pascal
procedure HandleWebkitInitialized;
var
  HookConsole : string;
begin
  HookConsole:='(function() {' +
                        ' var oldlog;' +
                        ' if (!console) console = { }; '+
                        ' if (console.log) oldlog=console.log; '+
                        ' console.log = function() {' +
                        '             if (oldlog) oldlog.apply(window,arguments); '+
                        '             indow.'+ SFuncDoLog+'.apply(window,arguments); '+
                        ' };' +
                        '})();';
  CefRegisterExtension('v8/hookconsole', HookConsole,Nil);
end;
```

People familiar with Javascript will see that
this little piece of Javascript defines
`console.log` if it does not yet exist, or
reroutes it to our `appLog` function.

❻ GETTING NOTIFIED OF EXCEPTIONS
We want to be notified whenever the Javascript
code throws an exception that is not explicitly
handled. This can be done in the
OnUncaughtException event, which we set to
the HandleJSException method.
This method is quite simple. It converts the
exception message to a string and sends a log
message to the BROWSER process:

```pascal
procedure HandleJSException(const browser: ICefBrowser;
                            const frame: ICefFrame;
                            const context: ICefv8Context;
                            const exception: ICefV8Exception;
                            const stackTrace: ICefV8StackTrace);
Var
  msg: ICefProcessMessage;
begin
  msg := TCefProcessMessageRef.New(SMsgDisplayLogMessage);
  msg.ArgumentList.SetString(0,'Exception :'+Exception.Message);
  Browser.MainFrame.SendProcessMessage(PID_BROWSER, msg);
end;
```

We could also add a stacktrace, but for
simplicity we limited ourselves to simply
logging the exception message. Note that
this method received the actual browser
instance: we use that to send the message
to the BROWSER process.

❼ COMMUNICATION FROM BROWSER TO RENDER PROCESS
We've shown how to create Javascript
objects and functions, and in the code
handling these functions we've seen how
messages are sent from the RENDER
process to the BROWSER
process.

When the user clicks a menu item in
the tray icon's popup menu, the
BROWSER process needs to
communicate this back to the RENDER
process, and specifically to the
Javascript handler that was passed as
part of the creation of the menu item.
This is where the Chromium app's
event handler
`OnProcessMessageReceived` comes
in: this event is triggered when the
RENDER process receives a message
from the BROWSER process. We set
the event to the `HandleProcessMessage`
procedure:

```pascal
procedure HandleProcessMessage(const browser: ICefBrowser;
                const frame: ICefFrame;
                sourceProcess: TCefProcessId;
                const aMessage: ICefProcessMessage;
                var aHandled: boolean);
var
  aID : Integer;
begin
  if (aMessage.name=SMsgExecuteTrayMenuClick)
      and (aMessage.ArgumentList.GetSize>0) then
  begin
    aID:=aMessage.ArgumentList.GetInt(0);
    TTrayIconHandler.CallTrayMenuCallBack(aID);
    aHandled:=true;
  end;
end;
```

The `CallTrayMenuCallBack` method is responsible for calling the Javascript callback that was registered together with the menu item. It's again quite simple:

```
class procedure TTrayIconHandler.CallTrayMenuCallBack(aID: integer);
Var
  M : TTrayMenuItem;
  arguments: TCefv8ValueArray;
begin
  If not Assigned(FMenu)then exit;
  M:= TTrayMenuItem(FMenu.FindItemID(aID));
  if (M=Nil) then
  SetLength(arguments,1);
  Arguments[0]:=TCefv8ValueRef.NewInt(aID);
  M.Handler.ExecuteFunctionWithContext(M.Context, nil, arguments);
end;
```

Since we need to be informed when the javascript logs a message or wants to create or remove a menu item, we set this event in the form's `OnCreate` event:

Here we can see that the `M.Handler` property, a function value (which is of type `ICefv8Value`) has a `ExecuteFunctionWithContext` method:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  TLocalResourceHandler.BaseDir:=ExtractFilePath(Paramstr(0));
  TLocalResourceHandler.RegisterHandler('local');
  bwBrowser.Chromium.OnProcessMessageReceived:=@HandleProcessMessage;
  bwBrowser.LoadURL(STrayIconURL);
end;
```

this will execute the function in the Javascript context which was captured when the handler was registered.
We must pass to this function the ID of the menu item, and this is done by creating an array of `iCEFv8value` interfaces. This array will be available in the Javascript function as the arguments variable.

The implementation of this message is again quite simple, and looks exactly the same as its counterpart in the RENDER process: (see next page)

❽ IMPLEMENTING THE GUI
All code presented till now will run in the RENDER process: the process where the HTML is displayed and Javascript is running. We now turn to the `BROWSER` process. This is the GUI application that we implement in Lazarus, the LCL. This is also where the tray icon and its associated popup menu is handled, and where the memo with the log messages is handled: the main form of the application.
The application is similar to the application developed in the previous article.
The `bwBrowser` component we used to display the browser window has a `Chromium` property, representing the `TChromium` instance responsible for the browser window.
The `TChromium` class has a `OnProcessMessageReceived` event. This event is triggered whenever the RENDER process sends a message to the BROWSER process.

```
procedure TMainForm.HandleProcessMessage(Sender: TObject;
const browser: ICefBrowser;
   const frame: ICefFrame;
   sourceProcess: TCefProcessId;
   const aMessage: ICefProcessMessage;
   out Result: Boolean);
Var
   aList : ICEFListValue ;
begin
   aList:=Nil;
   if Assigned(aMessage) then aList:=aMessage.ArgumentList;
   if not (Assigned(aList) and (aList.getSize>0)) then exit;
   Case aMessage.Name of
      SMsgSetTrayIconVisibility: Application.QueueAsyncCall(@SetTrayIconVisibility, Ord(aList.getBool(0)));
      SMsgAddTrayMenuItem:
            if (aList.getSize>1) then AddTrayMenuItem(aList.getInt(0), utf8Encode(aList.getString(1)));
      SMsgRemoveTrayMenuItem: RemoveTrayMenuItem(aList.getInt(0));
      SMsgDisplayLogmessage: DoLog('Render process: '+UTF8Encode(aList.getString(0)));
   end;
end;
```

As you can see, we capture 4 messages: the 4 different messages that we implemented in the RENDER process. Note the use of UTF8Encode: strings in the CEF4Delphi APIs are UTF16-encoded unicode strings. Lazarus uses UTF8-encoded strings, so to avoid compiler warnings, we manually convert them. The first method sets the visibility of the tray icon. It is wrapped in a `QueueAsyncCall` method, because as we've seen in the previous article, the messages can arrive in another thread than the main thread. The implementation is quite simple:

```
procedure TMainForm.SetTrayIconVisibility(
        aVisible : PtrInt);
begin
   TIApp.Visible:=aVisible<>0;
end;
```

The message signaling that a menu item must be added to the Tray Icon's popup menu is a little more complicated. It gets 2 parameters: the ID identifying the menu item, and the caption:

```
procedure TMainForm.AddTrayMenuItem(aAPIID : integer; ACaption : String);
Var
   tm : TTrayMenuItem;
begin
   Itm:=TTrayMenuItem.Create(Self);
   Itm.APIID:=aAPIID;
   Itm.Caption:=aCaption;
   Itm.OnClick:=@HandleTrayIconClick;
   PMTray.Items.Add(Itm);
end;
```

As you can see, the code creates a `TTrayMenuItem` menu item, it has an extra property `APIID` which holds the ID that was allocated for the menu item in the RENDER proces.
The method to remove an item is similar, and uses the `APIID` to find the menu item that must be removed:

```
procedure TMainForm.RemoveTrayMenuItem(aAPIID : integer);
Var
   Itm : TTrayMenuItem; I : Integer;
begin
   Itm:=Nil;
   I:=PMTray.Items.Count;
   While (I>=0) and (Itm=Nil) do
   begin
      Itm:=TTrayMenuItem(PMTray.Items[i]);
      if (Itm.APIID<>aAPIID) then
      Itm:=Nil;
      Dec(I);
   end;
   Itm.Free;
end;
```

The last message is the logging message. It uses a form variable `FLogMsg` to store the log message, and then uses the `QueueAsyncCall` mechanism to actually display the log message in `ShowLogMsg`:

```pascal
procedure TMainForm.DoLog(const aFmt: String; aArgs: array of const);
begin
  DoLog(Format(aFmt,aArgs))
end;

procedure TMainForm.DoLog(const aMsg: String);
begin
  FLogMsg:=aMsg;
  Application.QueueAsyncCall(@ShowLogMsg,0);
end;

procedure TMainForm.ShowLogMsg(Data: PtrInt);
begin
  MLog.Lines.Add(FLogMsg);
end;
```

```pascal
procedure TMainForm.HandleTrayIconClick(Sender: TObject);
var
  TempMsg : ICefProcessMessage;
  MID : Integer;
begin
  MID:=(Sender as TTRayMenuItem).APIID;
  DoLog('BROWSER process: click for tray menu item %d',[MID]);
  TempMsg:=TCefProcessMessageRef.New(SMsgExecuteTrayMenuClick);
  if TempMsg.ArgumentList.SetInt(0,MID ) then
    bwBrowser.Chromium.SendProcessMessage(PID_RENDERER, TempMsg);
end;
```

There is one more method that must be explained. The menu items created in the **AddTrayMenuItem** method had their **OnClick** handler set to the **HandleTrayIconClick** method. This method needs to send a message to the RENDER process to signal the click.
The same mechanism as in the RENDER process is used: create a message object, attach the ID to its list of arguments, and use the **SendProcessMessage** method of **bwBrowser.Chromium** to actually send the message (see coding left):

This message will then be handled by the **HandleProcessMessage** routine of the RENDER process.

❾ USING THE API IN JAVASCRIPT

After all this coding, it's time to show the fruits of all this labour: actually use the created classes in Javascript. The following very simple HTML is used:

```html
<!DOCTYPE html>
<html>
<head>
<title>Tray icon demo</title>
<link rel="stylesheet" href="notyf.min.css">
<script src="notyf.min.js"></script>
<script src="app.js"></script>
</head>
<body>
<h1>Tray icon.</h1>
<p>The following buttons set the visibility of
the tray icon: </p>
<button
onclick="makeVisible()">Visible</button>
<button
onclick="makeInVisible()">Invisible</button>
<p>Enter a caption and click the button to add
an entry to the tray menu:</p>
<input type="text" id="myCaption" value="">
<button onclick="addMenuItem()">Add menu
item</button><br>
<input type="text" id="myRemoveId" value="">
<button onclick="removeMenuItem()">Remove menu
item</button>
<p>After adding a menu item, you can right-
click on the menu item,
and the callback will be executed.</p>
<p>Click count: <span id="count">0</span>,
last ID: <span id="lastid">?</span></p>
</body>
```

the **notyf.min.js** script and associated CSS are a minimalistic message toast implementation.
It can be found on
https://carlosroso.com/notyf/
and it is used in the actual Javascript for our little HTML page, in the **app.js** file:

```javascript
var
  aCount = 0;
  notyf = new Notyf({position: {x:"center",y:"top"}});

function doLog(msg) {
  console.log(msg);
  notyf.success(msg);
}

function makeVisible() {
  window.trayIcon.visible=true;
  doLog('TrayIcon: '+window.trayIcon.visible);
}

function trayIconClicked(aID) {
  aCount=aCount+1;
  doLog("Menu item "+aID+" Clicked!");
  document.getElementById("count").innerText=aCount;
  document.getElementById("lastid").innerText=aID;
}

function makeInVisible() {
  window.trayIcon.visible=false;
  doLog('TrayIcon : '+window.trayIcon.visible);
}
```

```
function addMenuItem() {
var aCaption = document.getElementById("myCaption").value;
var aID =
window.trayIcon.addMenuItem(aCaption,trayIconClicked);
doLog("Menu item with ID "+aID+" Created!");
}

function removeMenuItem() {
var aID = document.getElementById("myRemoveId").value;
window.trayIcon.removeMenuItem(parseInt(aID,10));
doLog("Menu item with ID "+aID+" Created!");
}
```

The `console.log` messages will end up in our
memo, and the results of calls will be
shown with a Notyf sucesss toast as well.
The result of all this work can be admired in
the screenshot in figure 2 on page 13.

⑩ USING PAS2JS

Obviously, we would prefer to make the Javascript program as a Pascal program. This is also possible. We need to change the HTML a little bit for this: we must remove the onclick attributes in the button tags and we must make sure all buttons have an ID, so we can attach an onclick handler in code. Like all pas2js programs, we must call `rt1.run()` in the HTML file.
The result of such changes is the following HTML:

```html
<!DOCTYPE html>
<html>
<head>
<title>Tray icon demo</title>
<link rel="stylesheet" href="notyf.min.css">
<script src="notyf.min.js"></script>
<script src="trayicon.js"></script>
</head>
<body>
<h1>Tray icon.</h1>

<p>The following buttons set the visibility of the tray icon: </p>
<button id="btnVisible">Visible</button>
<button id="btnInvisible">Invisible</button>

<p>Enter a caption and click the button to add an entry to the tray
menu:</p>
<input type="text" id="myCaption" value="">
<button id="btnAddMenuItem">Add menu item</button><br>
<input type="text" id="myRemoveId" value="">
<button id="btnRemoveMenuItem">Remove menu item</button>
<p>After adding a menu item, you can right-click on the
menu item, and the callback will be executed.</p>
<p>Click count: <span id="count">0</span>,
last ID: <span id="lastid">?</span></p>

<script>
 rtl.run();
</script>
```

To be able to use the tray icon in pas2js, we must declare an external class: this allows the compiler to check the validity of the code. As a side effect, the Lazarus code completion will also work.
The definition is as follows:

```pascal
Type
  TClickHandler = reference to procedure (aID : Integer);

  TTrayIcon = class external name 'Object' (TJSObject)
  visible : boolean;
  function addMenuItem(aCaption :string;
                       aHandler : TClickHandler) : integer;
  procedure removeMenuItem(aID: Integer);
end;

Var
  trayIcon : TTrayIcon; external name 'window.trayIcon';
```

With this declaration, the compiler knows that there is a `TTrayIcon` class, and that there is an instance in `window.trayIcon`.
Something similar must be made for the `Notyf` class, this is done in the unit `libnotyf`.
This unit is now part of pas2js.
The `DoLog` procedure can now be written as:

```pascal
var
  notyf : TJSNotyf;
procedure DoLog(Msg : string);
begin
  console.log(msg);
  notyf.success(msg);
end;
```

The instance of `notyf` is create in the program main code block. We must also attach an `onclick` handler to the buttons in our HTML, because the HTML cannot refer to the pas2js methods. For this reason we had to give an id to each button in the HTML.
Attaching the `onClick` handler is also done in the program initialization code:

```pascal
begin
  opts:=TJSNotyfOptions.new;
  opts.position:=TJSNotyfPosition.New;
  opts.position.x:='center';
  opts.position.y:='top';
  notyf:=TJSNotyf.new(opts);
  AddClick('btnVisible',@DoVisibleClick);
  AddClick('btnInvisible',@DoInVisibleClick);
  AddClick('btnAddMenuItem',@DoAddMenuItem);
  AddClick('btnRemoveMenuItem',@DoRemoveMenuItem);
end.
```

As you can see, the creation of the `notyf` instance is done using pascal classes only: the compiler will check your code and guarantees a correctly constructed `Notyf` instance.
The `AddClick` routine is a small helper routine which makes the adding of the onclick handler a little more readable. The code of this routine is quite simple:

```pascal
Procedure AddClick(aName: string; aHandler : TJSRawEventHandler);
Var El : TJSHTMLElement;
begin
  El:=TJSHTMLElement(document.getElementById(aName));
  El.AddEventListener('click',aHandler);
end;
```

The event handlers to make the tray icon visible
or invisible are very simple:

```pascal
Procedure DoVisibleClick(aEvent: TJSEvent);
begin
  trayIcon.visible:=true;
  doLog('TrayIcon:'+BoolToStr(trayIcon.visible));
end;

Procedure DoInVisibleClick(aEvent: TJSEvent);
begin
  trayIcon.visible:=false;
  doLog('TrayIcon:'+BoolToStr(trayIcon.visible));
end;
```

```pascal
procedure trayIconClicked(aID : integer);
begin
  aCount:=aCount+1;
  doLog('Menu item '+IntToStr(aID)+' Clicked!');
  TJSHTMLElement(Document.getElementById
              ('count')).innerText:=IntToStr(aCount);
  TJSHTMLElement(document.getElementById
              ('lastid')).innerText:=IntToStr(aID);
end;
```

And that's all there is to it. The code is a little
more verbose than the Javascript code, but
it is type-safe and the compiler has verified that
all your code is syntactically correct, and
that the functions have the correct signatures
and get the correct amount of parameters.
The resulting program will of course behave
exactly the same as the Javascript version.

Again, the compiler will check your code.
The code to add and remove a menu item are
equally short. For readability, the code to get
the value of an input element has been split out
to a `getValue` function:

```pascal
function getValue(aID : String) : string;
var
  EL : TJSElement;
begin
  El:=document.getElementById(aID);
  Result:=JSHTMLInputElement(el).Value;
end;

procedure DoAddMenuItem(aEvent : TJSEvent);
Var
  aCaption : String;
  aID : Integer;
begin
  aCaption:=getValue('myCaption');
  aID:=trayIcon.addMenuItem(aCaption,@trayIconClicked);
  DoLog('Menu item with ID '+IntToStr(aID)+' Created!');
end;

procedure DoRemoveMenuItem(aEvent : TJSEvent);
var
  aID : Integer;
begin
  aID:=StrToIntDef(GetValue('myRemoveId'),-1);
  if aID<>-1 then
  trayIcon.removeMenuItem(aID);
  DoLog('Menu item with ID '+IntToStr(aID)+' removed!');
end;
```

The main difference of this code with the
Corresponding Javascript code is that it is
typesafe. The last routine is the `onclick` handler
of the menu item, which we called
`trayIconClicked`:

⑪ CONCLUSION

To add objects to a Javascript environment in
CEF is possible, and opens a lot of possibilities
for interacting with the OS. It allows you to
create an Electron-like environment - much as
the 'Miletus' product from TMS Software does.
The mechanisms are a little cumbersome, but it
should be possible to reduce the amount of
needed code by making clever use of RTTI:
leveraging (extended) RTTI, it should be possible
to directly expose a Pascal class in the Javascript
environment, without having to write all this glue
code. This we will investigate in a future
contribution.

Lazarus: New free TMS Webcore for Lazarus
By DetlefOverbeek & Coding Mattias Gärtner

ANNOUNCEMENT:
new free TMS Webcore for Lazarus
which means you can freely try but do not have the sourcecode available of the components.
I have created a very small project to test on all versions of Lazarus: Mac, Linux and Windows.
So because of this I can tell you the macOS version has been tested with the newest macOS Big Sur, Lazarus 2.012.
The Linux version is Mint, Lazarus 2.012 and works fine like the Windows 10 Version.
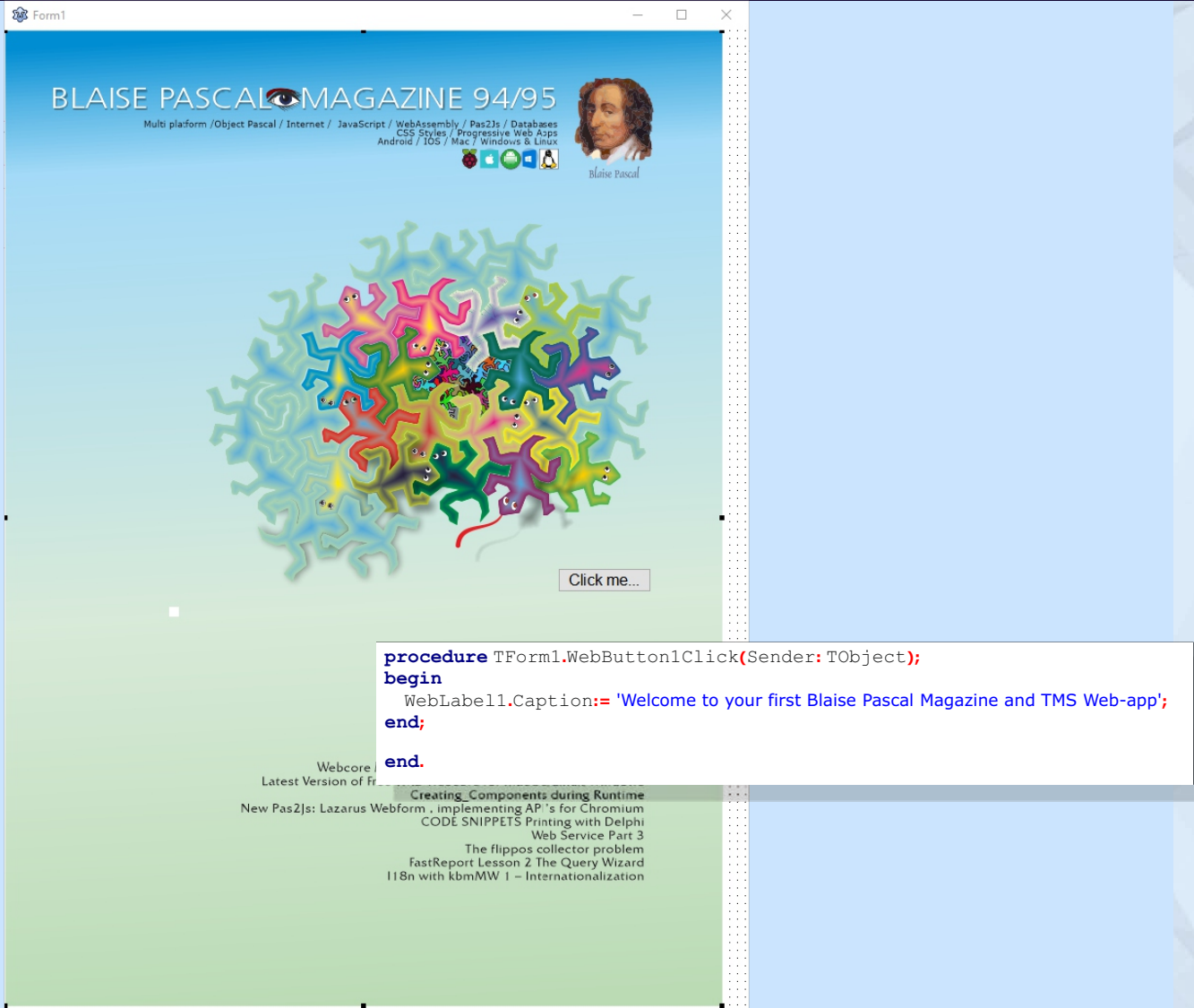So now you have the ability to test Webcore from TMS as much as you want. Play with it!
You can download it from your site:
`https://www.blaisepascalmagazine.eu/your-downloads/`
The project is available there as well.
You need to get a FREE Registration Key from TMS:
`https://www.tmssoftware.com/site/trialkey.asp`


```pascal
procedure TForm1.WebButton1Click(Sender: TObject);
begin
  WebLabel1.Caption:= 'Welcome to your first Blaise Pascal Magazine and TMS Web-app';
end;

end.
```

BASIC WIZARDS
FOR BUILDING REPORTS
INTRODUCTION

In this second part I will explain the use of a database, SQL and reporting. The programs used for these examples can be obtained from your downloadpage(after logging in).

To start with this project you first need to start your Delphi version and the installed version of FastReport completed. After starting you will see at the right hand an overview of the possible databases. We chose to use (see the Data Explorer) and then DBDemos, which is an Access Database from Microsoft. As soon you have made the correct settings you will be able to view the Tables. So simply follow the settings on the next pages we have prepared for you. The necessary settings are circumcised in red.

We have used as you can see in the Data Explorer (which is devided in two different segments: **FireDAC** and **dbExpress)**
I use Firedac because I think this version is the best.
Please choose:
1. Microsoft Access Database
2. Choose DBDEMOS and make sure that you can see in the list of tables and the fields.
The DBDEMOS database is included in the project. So the path to it is the same as your project.
There are all kinds of drivers, but don't get confused, we do not need that. So far the settings inside Delphi for the DBDDEMOS.
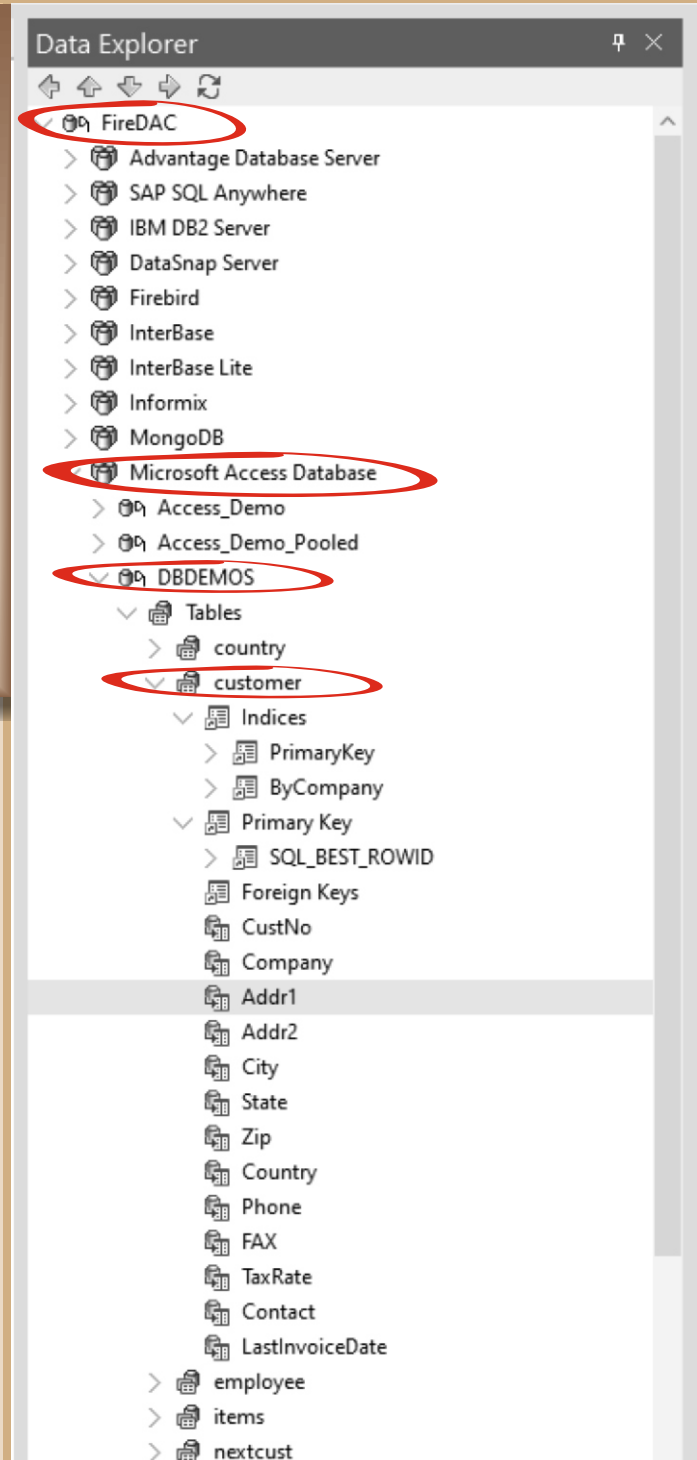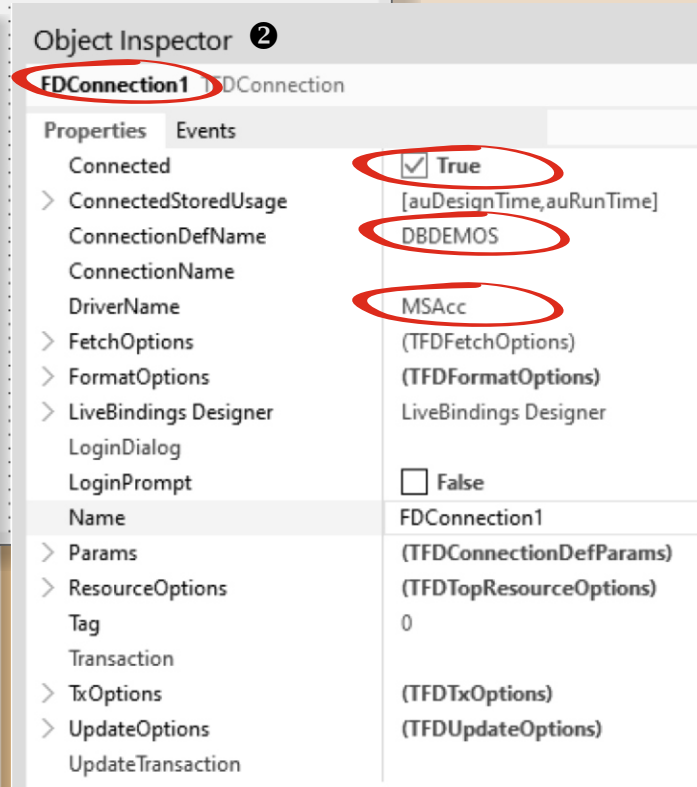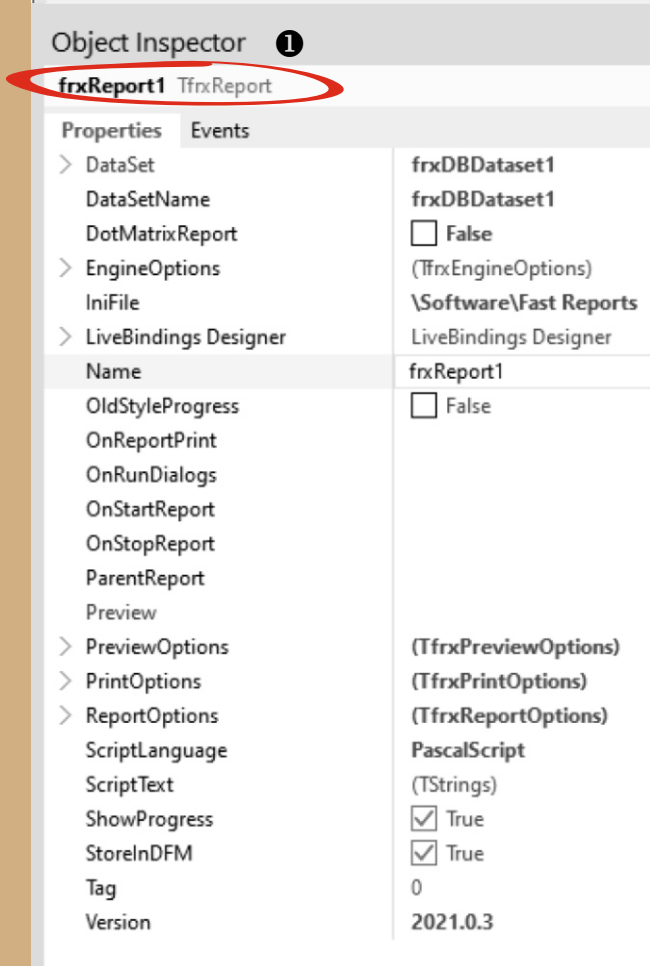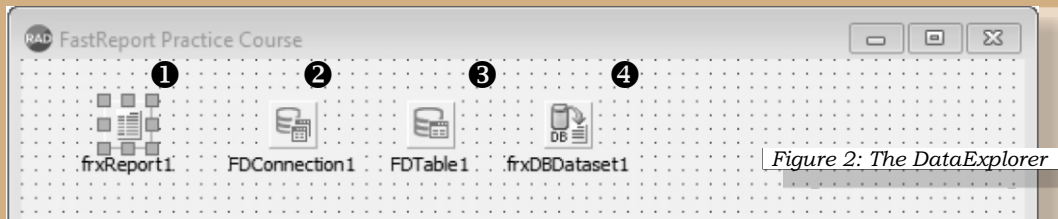

Figure 1: The DataExplorer

FastReport Practice Course

❶ frxReport1   ❷ FDConnection1   ❸ FDTable1   ❹ frxDBDataset1

Figure 2: The DataExplorer

Object Inspector ❶

frxReport1 TfrxReport

Properties	Events

> DataSet	frxDBDataset1
DataSetName	frxDBDataset1
DotMatrixReport	☐ False
> EngineOptions	(TfrxEngineOptions)
IniFile	\Software\Fast Reports
> LiveBindings Designer	LiveBindings Designer
Name	frxReport1
OldStyleProgress	☐ False
OnReportPrint	
OnRunDialogs	
OnStartReport	
OnStopReport	
ParentReport	
Preview	
> PreviewOptions	(TfrxPreviewOptions)
> PrintOptions	(TfrxPrintOptions)
> ReportOptions	(TfrxReportOptions)
ScriptLanguage	PascalScript
ScriptText	(TStrings)
ShowProgress	☑ True
StoreInDFM	☑ True
Tag	0
Version	2021.0.3

Figure 3: The settings fore the frxReport

Object Inspector ❷

FDConnection1 TFDConnection

Properties	Events

Connected	☑ True
> ConnectedStoredUsage	[auDesignTime,auRunTime]
ConnectionDefName	DBDEMOS
ConnectionName	
DriverName	MSAcc
> FetchOptions	(TFDFetchOptions)
> FormatOptions	(TFDFormatOptions)
> LiveBindings Designer	LiveBindings Designer
LoginDialog	
LoginPrompt	☐ False
Name	FDConnection1
> Params	(TFDConnectionDefParams)
> ResourceOptions	(TFDTopResourceOptions)
Tag	0
Transaction	
> TxOptions	(TFDTxOptions)
> UpdateOptions	(TFDUpdateOptions)
UpdateTransaction	

Figure 4:
The FDConnection has 3 important settings
that need to be made

We have created an overview of all the
different settings made inside Delphi:
the components
1: The frxReport
2: The FDConnection
3: The FDTable
4: The frxDBDataset1
As a result you can see the Report Editor and
the Project form with components.

Object Inspector ❸

FDTable1 TFDTable

Properties	Events

Active	☑ True
ActiveStoredUsage	[auDesignTime,auRunTime]
Aggregates	(TFDAggregates)
AggregatesActive	☐ False
AutoCalcFields	☑ True
CachedUpdates	☐ False
CatalogName	
ChangeAlerter	
ChangeAlertName	
Connection	FDConnection1
ConnectionName	
Constraints	(TCheckConstraints)
ConstraintsEnabled	☐ False
DetailFields	
Exclusive	☐ False
FetchOptions	(TFDFetchOptions)
FieldOptions	(TFieldOptions)
Filter	
FilterChanges	[rtModified,rtInserted,rtUnmodified]
Filtered	☑ True
FilterOptions	[]
FormatOptions	(TFDFormatOptions)
IndexFieldNames	CustNo
IndexName	
LiveBindings Designer	LiveBindings Designer
LocalSQL	
MasterFields	
MasterSource	
Name	FDTable1
ObjectView	☑ True
ResourceOptions	(TFDBottomResourceOptions)
SchemaAdapter	
SchemaName	
TableName	customer
Tag	0
Transaction	
UpdateObject	
UpdateOptions	(TFDBottomUpdateOptions)
UpdateTransaction	

Figure 5:
The FDTable settings, especially CustNo and Table name

Object Inspector ❹

frxDBDataset1 TfrxDBDataset

Properties	Events

BCDToCurrency	☐ False
CloseDataSource	☐ False
DataSet	FDTable1
DataSource	
Description	
Enabled	☑ True
FieldAliases	(TStrings)
IndexTag	0
LiveBindings Designer	LiveBindings Designer
Name	frxDBDataset1
OpenDataSource	☑ True
RangeBegin	rbFirst
RangeEnd	reLast
RangeEndCount	0
Tag	0
UserName	frxDBDataset1

Figure 6: The frxDBDataset

Structure

⬚ Form16
  - FDConnection1
  - FDTable1
    - Aggregates
    - Constraints
    - Fields
      - 0 - CustNo {FDTable1CustNo}
      - 1 - Company {FDTable1Company}
      - 2 - Addr1 {FDTable1Addr1}
      - 3 - Addr2 {FDTable1Addr2}
      - 4 - City {FDTable1City}
      - 5 - State {FDTable1State}
      - 6 - Zip {FDTable1Zip}
      - 7 - Country {FDTable1Country}
      - 8 - Phone {FDTable1Phone}
      - 9 - FAX {FDTable1FAX}
      - 10 - TaxRate {FDTable1TaxRate}
      - 11 - Contact {FDTable1Contact}
      - 12 - LastInvoiceDate {FDTable1LastInvoiceDate}
  - frxDBDataset1

Figure 7: The Structure overview shows what is available

Figure 7: The frxReport can be edited:

Right click and edit. If you choose to
simply open you can later use the the
opening of the wizard. See page 6.
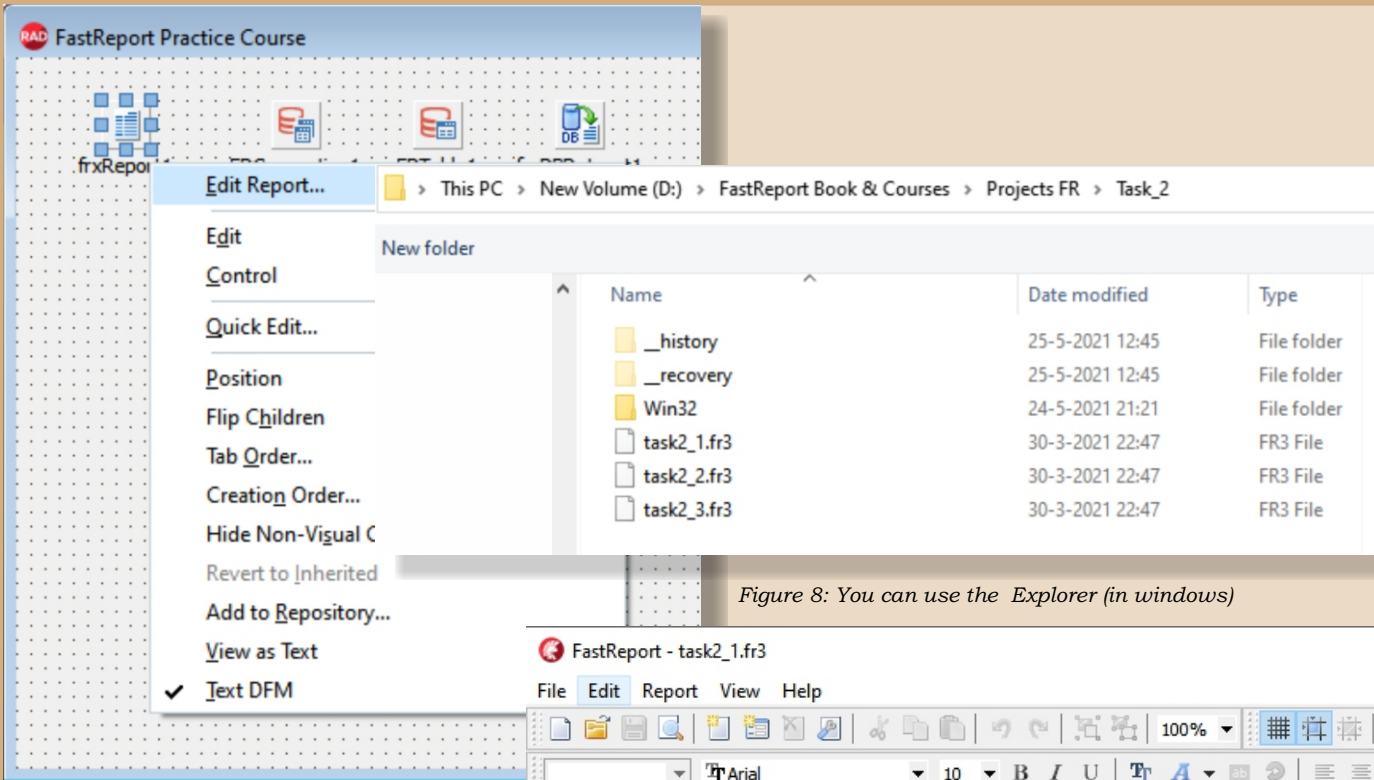
Figure 8: You can use the Explorer (in windows)
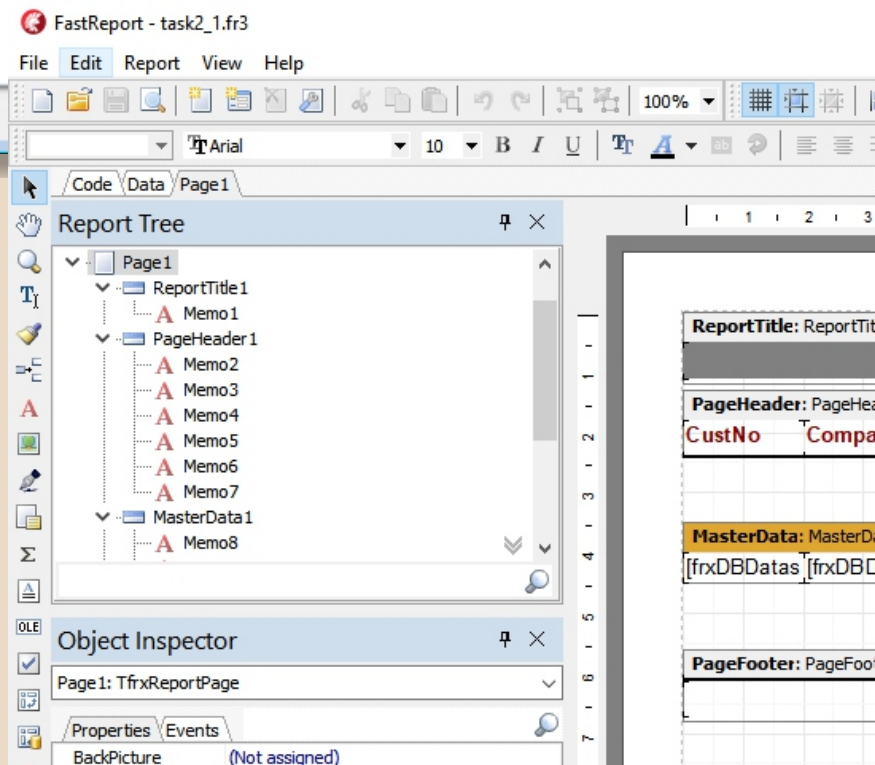
Figure 9: The Report Designer

If you right click on `File→New` you can start a
new report or you can choose one,
as we did here.
There is of course the option to open a report:
Choose the report called **task2_1.fr3.**
You will instantly get all the right settings for
the project. The other reports should not yet
be tried.
Within the **Report Editor** you now can go to:
➜ **File** ➜ **New …**
A Wizard will appear.
We are going to through all these options.

New...

New Report

New Page

New Dialog

Open...

Save

Save As...
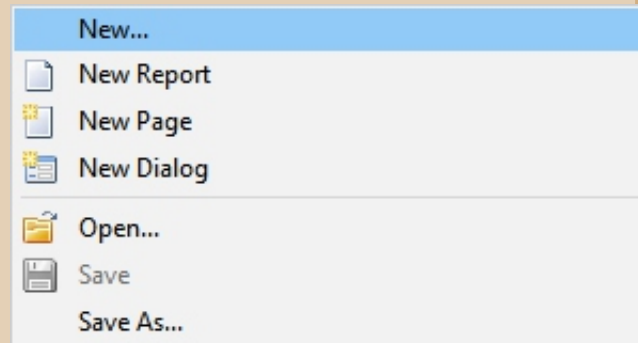
Figure 10 - Start window
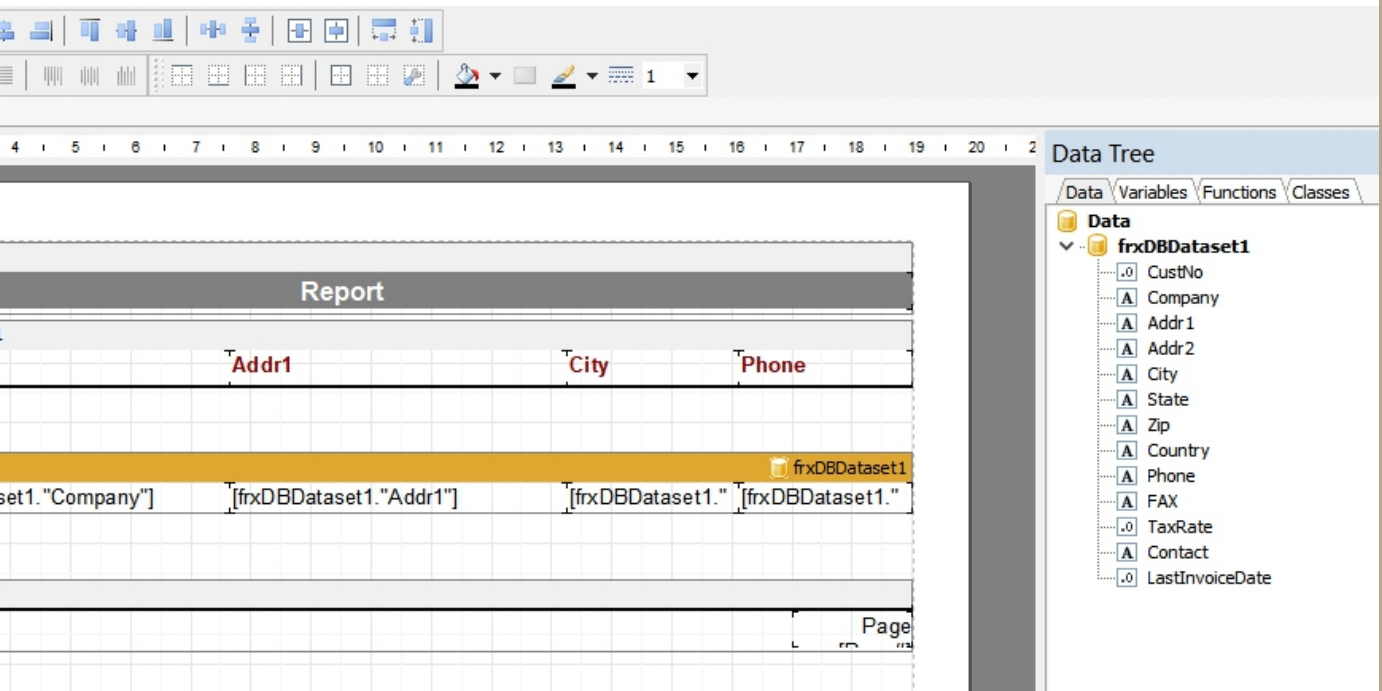
To make it easier for users to work in the **Report Designer**, it has been provided with helpers - masters. Such wizards allow you to specify the main parameters of the report, which will form a ready-made template.

Wizards can be distinguished by their intended purpose (See figure 11):
- The masters of the new report;
- Wizard for the new database connection;
- Master of the new table;
- The master of the new inquiry.
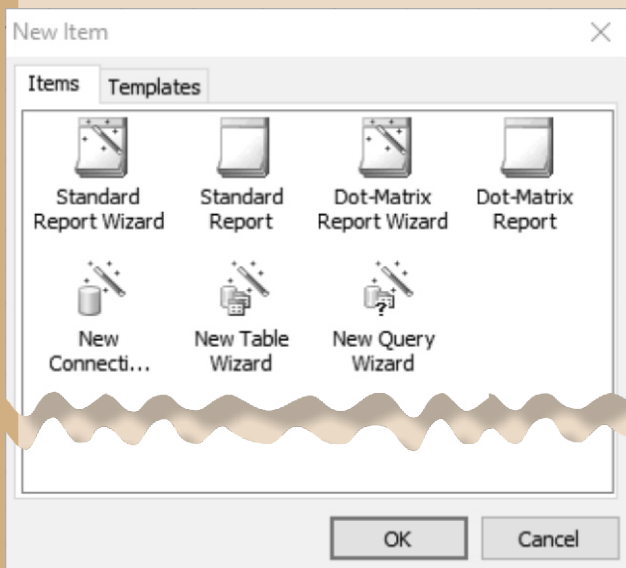
Figure 11 - Wizards

For the help in drawing up of inquiries to a database there is a special constructor of inquiries about which we will speak a little later.

MASTER OF NEW REPORT
The master of the new report is divided into four:
❶ The master of the standard report;
❷ **Matrix report master;**
❸ It's a blank standard report;
❹ **Empty matrix report.**

The **"Blank Standard Report"** and **"Blank Matrix Report"** wizards generate a blank report (*for standard or matrix printers - you can read about the matrix reports in the next chapter*), which contains one page.

The Wizards **"Standard Report Wizard"** and **"Matrix Report Wizard"** allow you to select the list of fields to be displayed in the report, grouping and method of placing the fields in the report.
Let's consider creating a report using the **"Standard Report Wizard"** in more detail.

Select the menu **"File | New ...",** in the opened window - item **"Standard Report Wizard"**.
We will see the window of the report wizard:
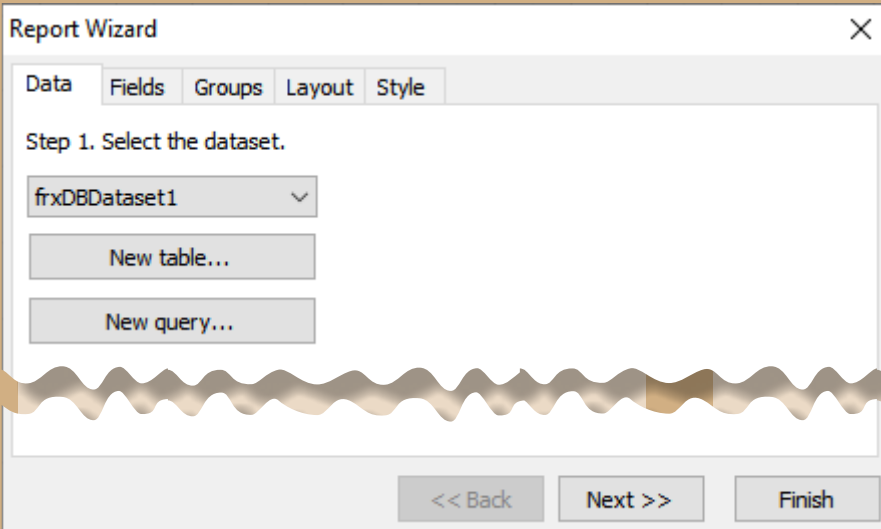
Figure 12 - Standard Report Wizard

As you can see, the window has several bookmarks. On the first tab, we need to select the data source on which the report will be built. To do this, first select the desired database connection. Select the data source - table and click the **"Next >>"** button.
On the next tab (Figure 13) you need to select the fields from the tables that you want to show in the report:

The list on the left shows the available fields, while the list on the right shows the selected ones. You can move fields from one list to another using the "Add", "Add all", "Delete all" buttons. Using the buttons of fields you can change places.
Let's add Company, Contact, Phone, FAX fields to the list of selected fields and press the **"Next >>"** button at the bottom of the figure 13.
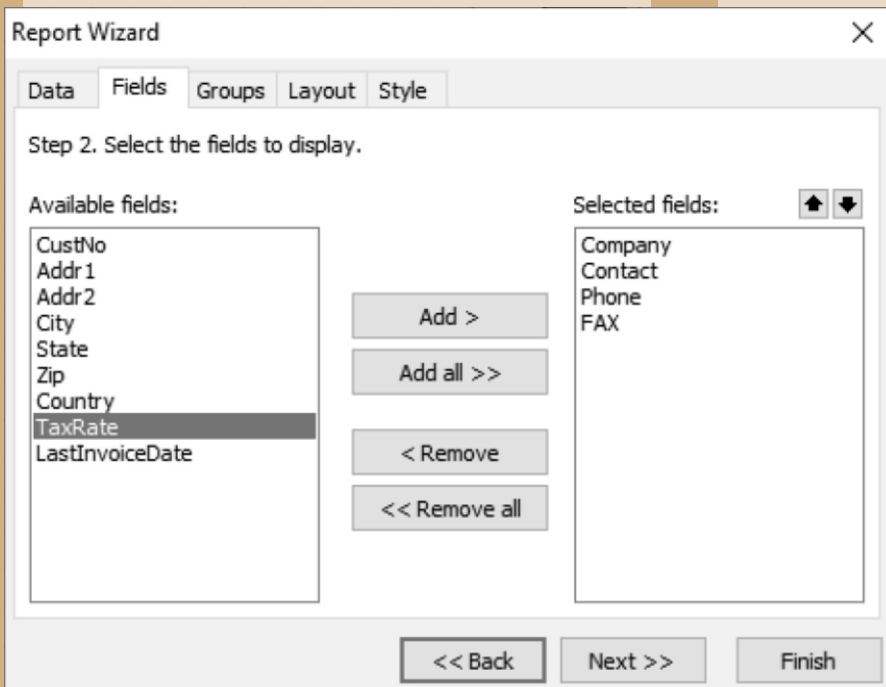
Figure 13:
Selection of table fields to be displayed in the report

Report Wizard

Data | Fields | Groups | Layout | Style

Step 3. Create groups (optional).

Available fields:

Company
Contact
Phone
FAX

Add >

< Remove

Groups:

<< Back | Next >> | Finish

Figure 14 - Selecting groups

On the next tab (Figure 14) you can add a grouping of one or more selected fields to the report. Bands Group header, Group footer will be added to the report.
You can skip this step - click **"Next >>"**.

On the next tab (Figure 15) you can choose the orientation of the report page and the way the fields are placed on it:

Report Wizard

Data | Fields | Groups | Layout | Style

Step 4. Define the page orientation and data layout.

Orientation
● Portrait
○ Landscape

Layout
● Tabular
○ Columnar

☑ Fit fields to page width

Report

Company	Address
Action Club	PO Box 5
Action Diver Supply	Blue Spar
Adventure Undersea	PO Box 7
American SCUBA	1739 Atla
Aquatic Drama	921 Everg
Blue Glass Happiness	6345 W. S

<< Back | Next >> | Finish

Figure 15- Page settings

You can choose the style of placement of the fields - tabular when the fields are located from left to right, or column when the fields are located under each other. When you select the field placement, the report image on the right side is redrawn. The "Fit all fields by width" option selects the width of the selected fields so that all fields fit on the page.

Finally, on the last tab (Figure 16), we can choose the report style - the color palette of the various report elements.
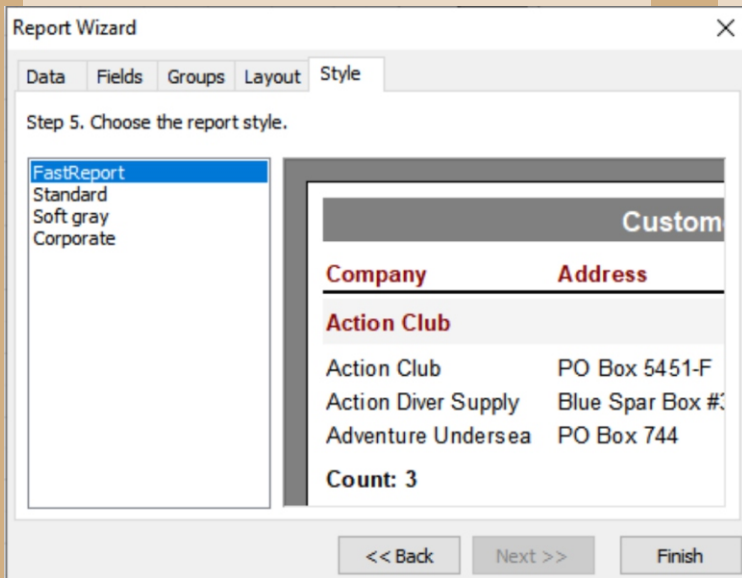
Figure 16 - Selection of report color palette

To finish the standard report wizard, click the **"Done"** button. After that the wizard will create the following report:

Figure 17 - Example of a report created using the wizard

You can view the report immediately in the preview window. Thus, using the wizard, we create a complete report that displays data from the database. For a novice user this feature will be very useful and will save time required for detailed familiarization with the **Designer.** Of course, you need to be able to build reports manually as well, but thanks to the wizard, you can simplify the process of learning the **Designer.**

NEW CONNECTION WIZARD
The **New Connection Wizard** allows you to add a new connection to the database to an already existing report. This may be necessary if you want to display data from two or more databases in the report. The wizard adds a component such as an **ADO** database to the report. In Figure 18, you can see the **Database Connection Wizard** window.
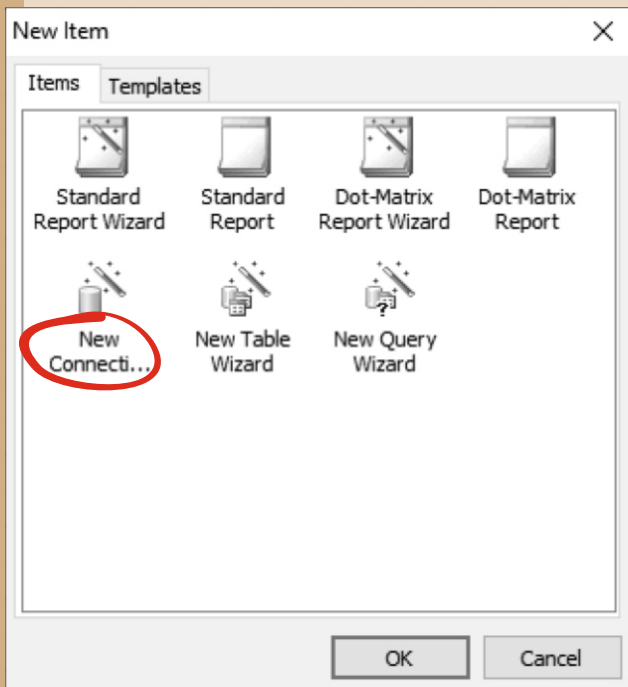
Figure 18 - New Connection Wizard
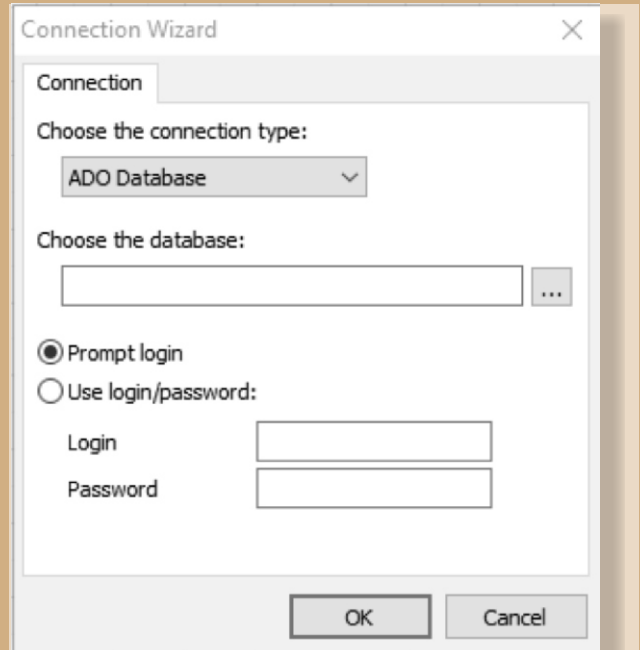
Figure 19 - The ADO Database Chosen

First, you need to select the type of connection. By default, **ADO** is selected, but **BDX** and **DBE** are also available.
Then, create a `connection string`, and the standard Windows window will open, where you can select the connection type and its parameters (Figure 19). After that, set a username and password, if necessary.
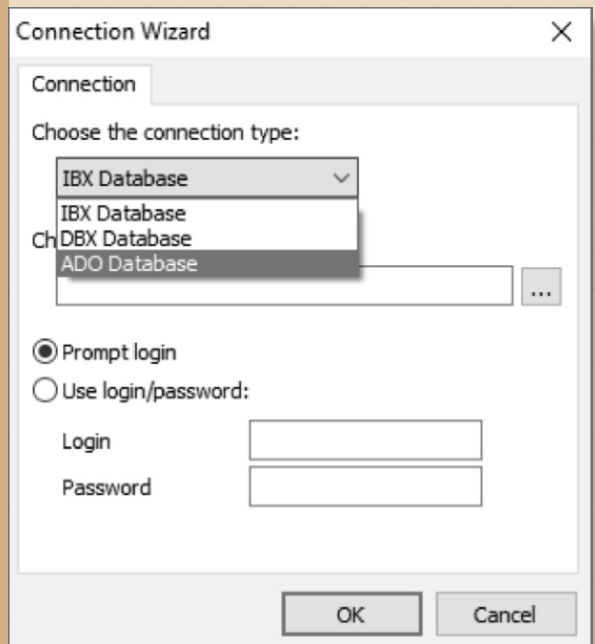
Figure 20 - The dropdown menu
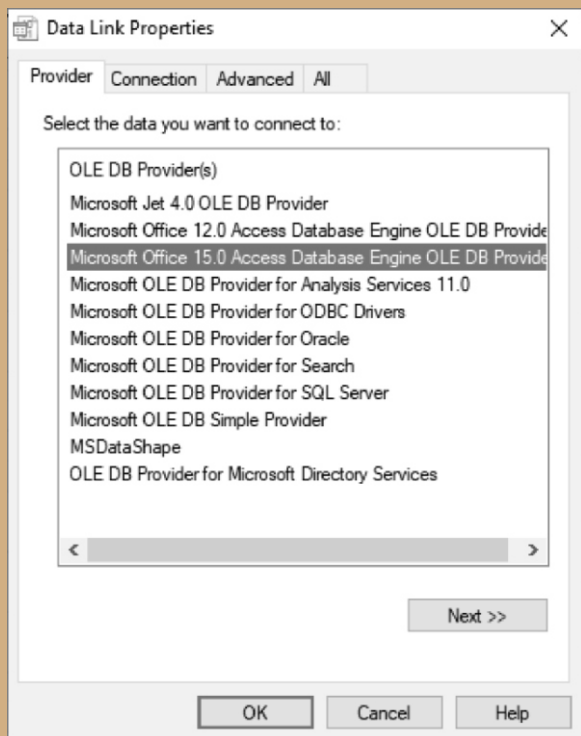
Figure 21 - Standard Windows window for connecting the database

Note: You can also create a new connection by switching to the **"Data"** tab and adding the **"ADO Database"** component to the report.
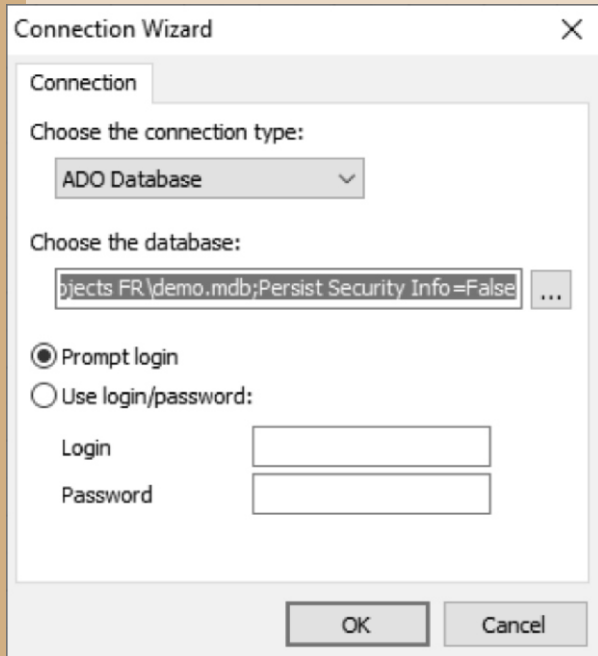
Figure 22 - Entering the path
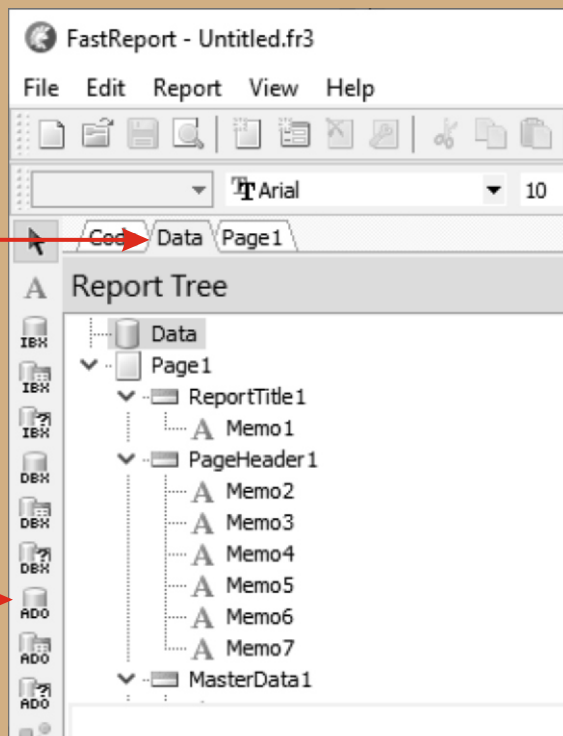
Figure 23
Add a new database by dropping the component

MASTER OF THE NEW TABLE
This wizard (Figure 24) allows you to add a new data source - a table - to an already existing report.
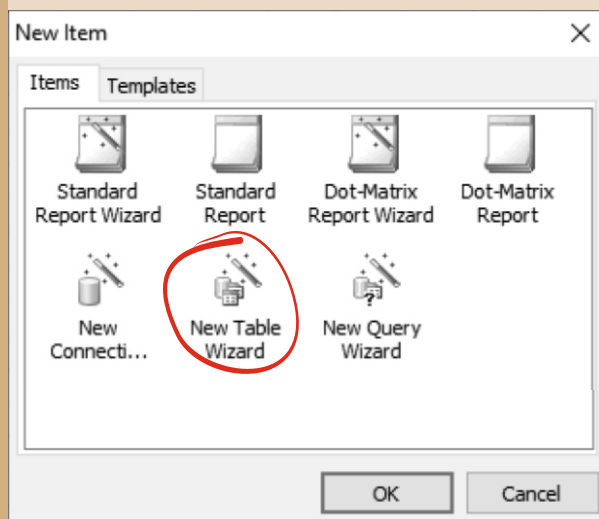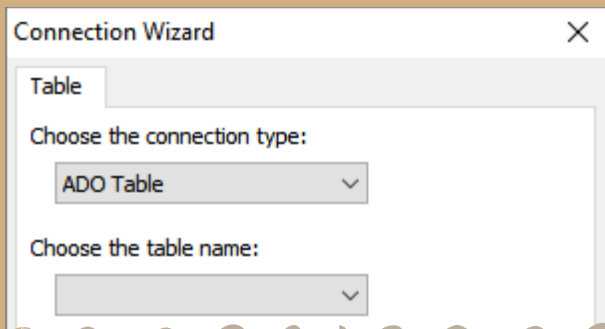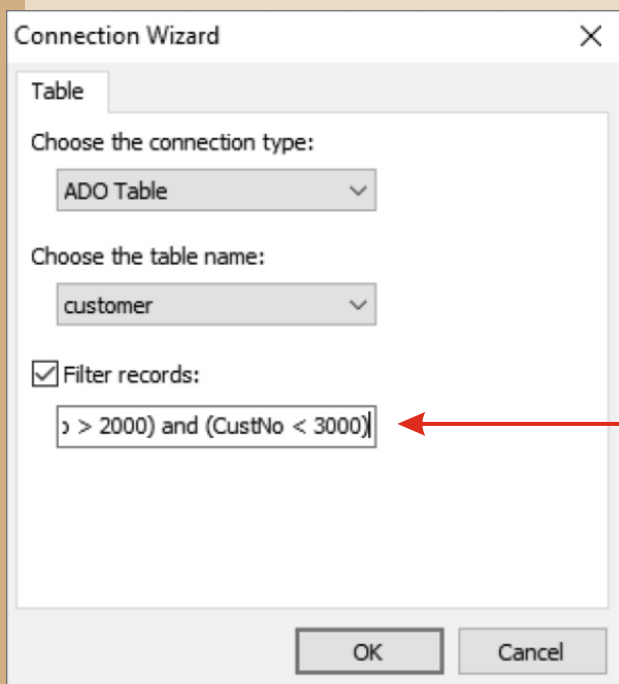

Figure 24 - Master of the new table

Figure 26 Choose Customer

Figure 27 - New Query wizard

Figure 26 - Master of the new table

In the wizard window you should select the table name. You can also specify a condition for filtering the table records, for example:

`(CustNo > 2000) and (CustNo < 3000)`

Note: You can also create a new table by switching to the **"Data"** tab and adding the **"ADO Table"** component to the report.

NEW QUERY WIZARD
The New Query Wizard (Figure 21) allows you to add a new data source to an existing report - an SQL query.

Figure 28: In the wizard window you should enter the SQL query text.

You can also use the visual query constructor by pressing the 🔲 icon.
The visual constructor will be described further in this chapter.

Note: You can also create a new query by switching to the "Data" tab and adding the "ADO query" component to the report.
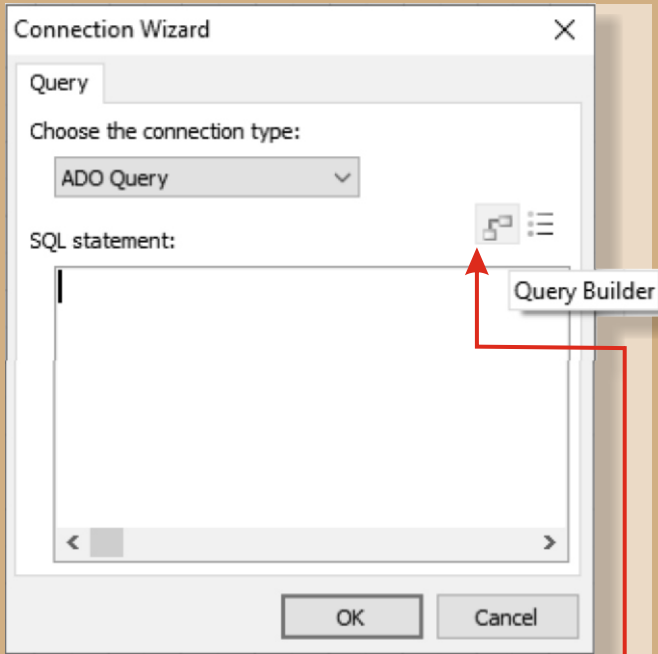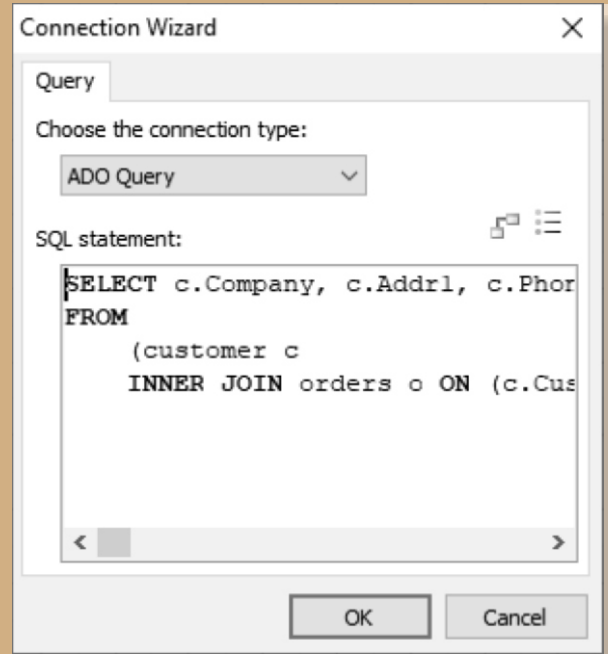
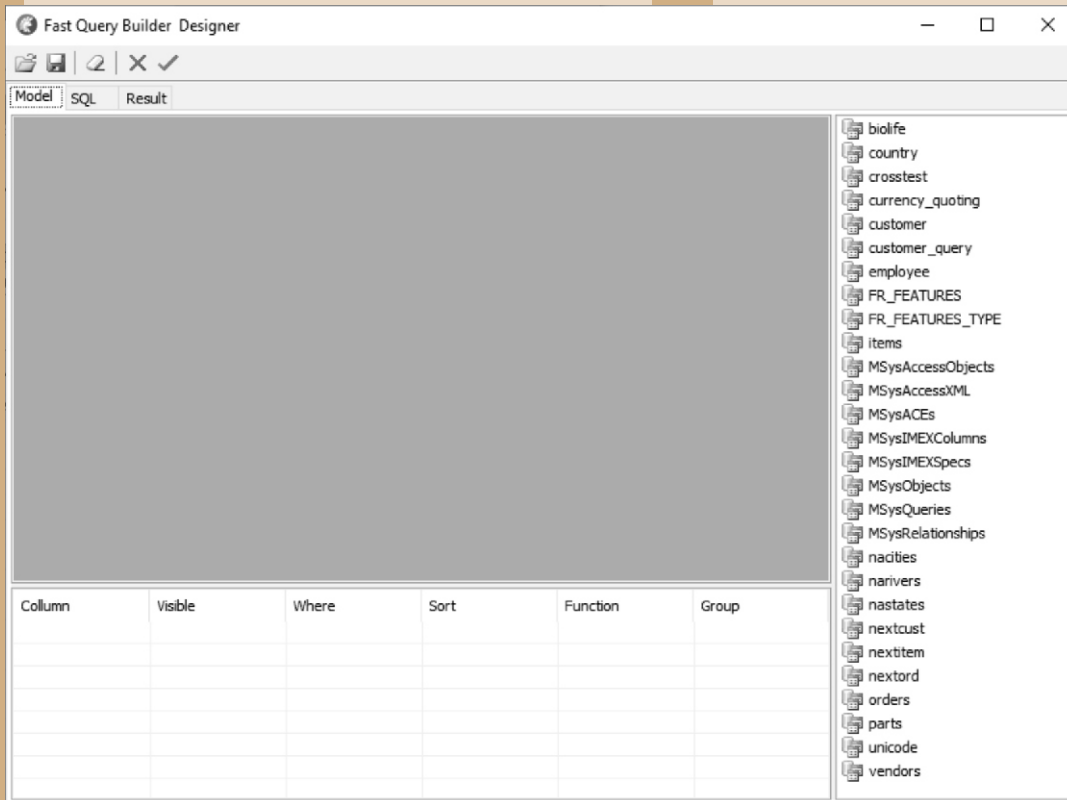Figure 29: Click on the Query Builder Icon

Figure 30: An Example

Figure 31: The Query Designer pops up

QUERY BUILDER DESIGNER
FastReport VCL (Professional, Enterprise versions) include a visual query builder. (FastQueryBuilder is used for this purpose and is also available as a separate product for use in your applications). The Query Builder is designed to build a query text in SQL using visual tools. Figure 32 shows the appearance of the constructor.
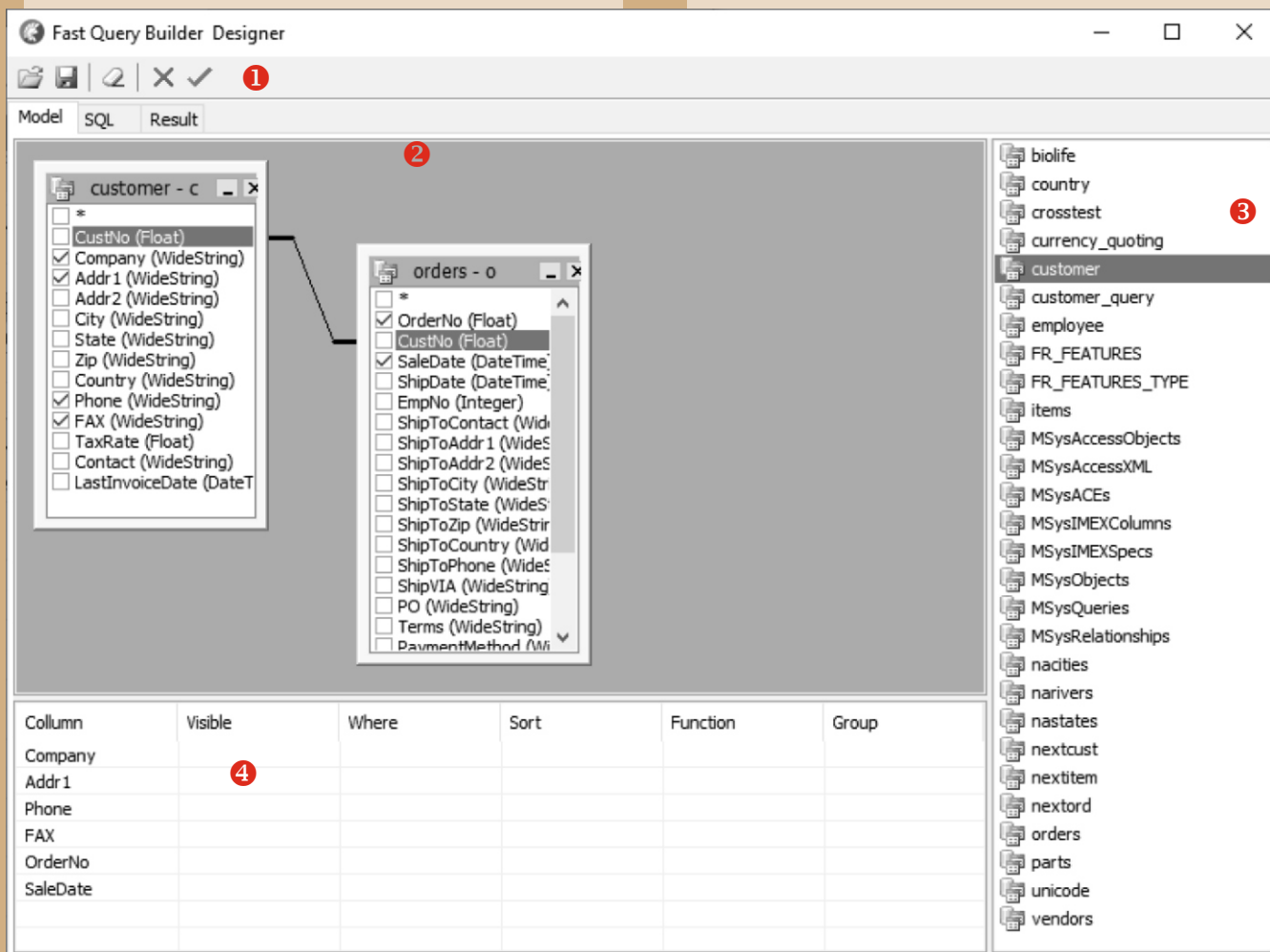

Figure 32 - Visual query builder.

The numbers on the picture are marked:
- ❶ toolbar,
- ❷ designer working field,
- ❸ list of available tables,
- ❹ parameters of selected table fields

As you can see by the figure, the visual constructor is divided into 3 zones: on the right - the list of tables, in the center - the working area, at the bottom - the window of table parameters.
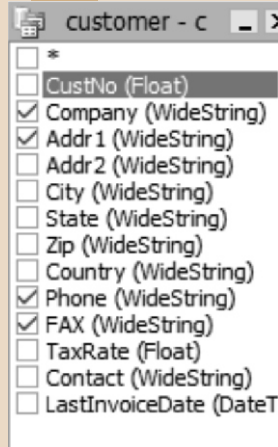
The toolbar is represented by the following elements:

- Open a SQL file
- Save the request to a file
  (the request scheme is also saved in a file)
- Designer workspace cleansing
- Button Okay.
  Exit the designer with saving.
- Cancel button.
  Exit the designer without saving

The constructor working field and the list of available tables support Drag&Drop technology, i.e. to place a table in the working area it is enough to move it there with the mouse. Another option is to double-click the table name in the list of available tables.

Let's look at the table in more detail. The table window contains all its fields.
You can mark the necessary fields to include them in the query (Figure 33).

Figure 33 - Table window

The marked fields will appear in the parameter window:

Figure 34 - Table parameter window.
See explanation below.

The following options are available for editing:
- ❶ Visibility - determines whether a field falls into the structure select
- ❷ Where is the condition for selecting a field. For example '> 5'.
- ❸ Sort - determines the sorting by field.
- ❹ Function - determines the function applicable to the field
- ❺ Group - grouping by field.
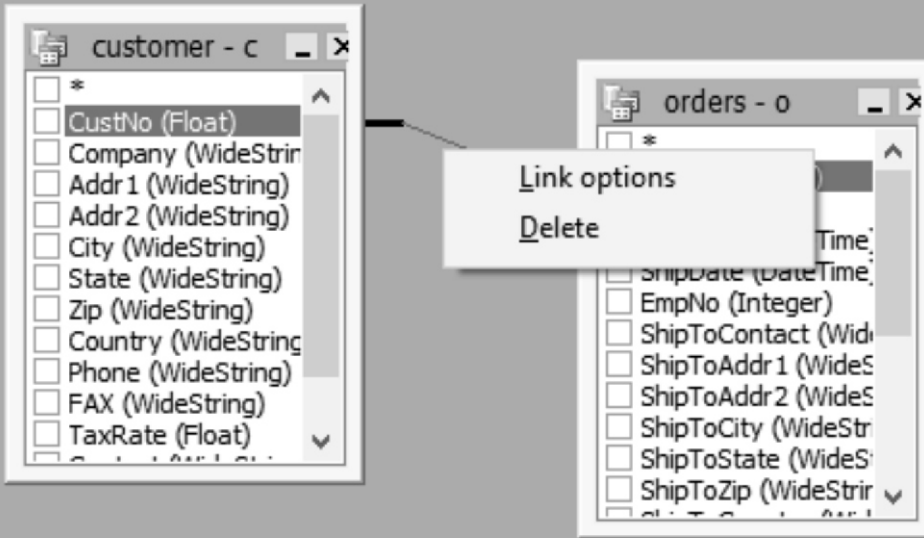
Figure 35: Select the communication line

When a connection is created, the compatibility of the field types is checked. You cannot create a connection between incompatible fields.

To set up the communication settings, click on the communication line and select Link options. See figure 35.
The communication parameter window will appear (Figure 36):
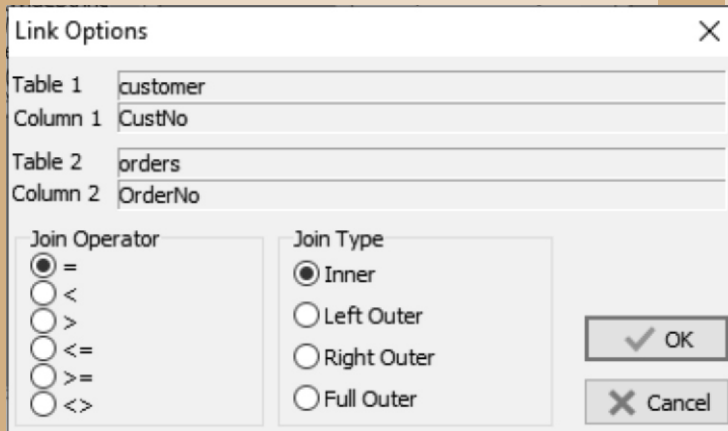
Figure 36: Link Options

VISUAL QUERY DESIGN (example)

Click the "New report" button on the designer toolbar. This will create a report page with the "Report title", "Level 1 data" and "Page Footer" legends.

Put the "ADO Query" component on the "Data" page. Double-click on the component and you will see the request editor window.
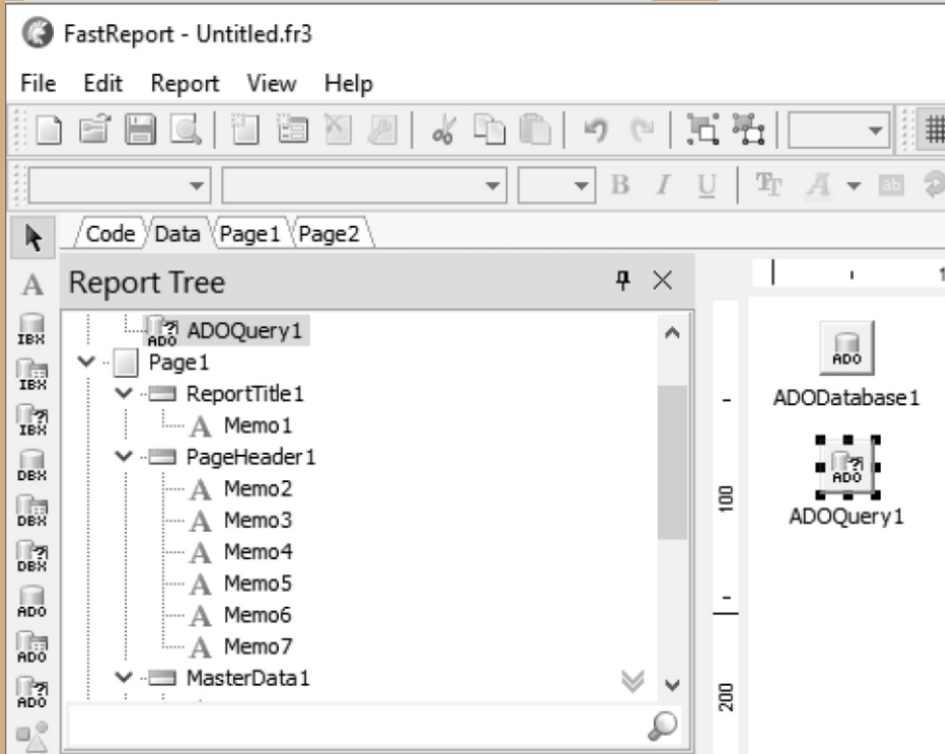
Figure 37 - Create a new report

Click the button in the editor and you will see the query constructor window. Select the Customer table in the left part of the window and move it to the working field (you can also double-click to move the table). Tick the CustNo, Company, Phone checkboxes:
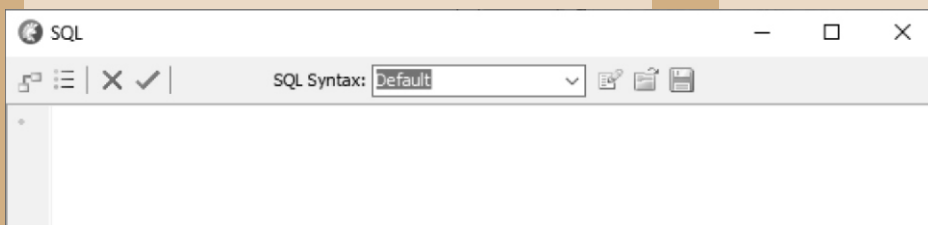
Figure 38 - The SQL Editor
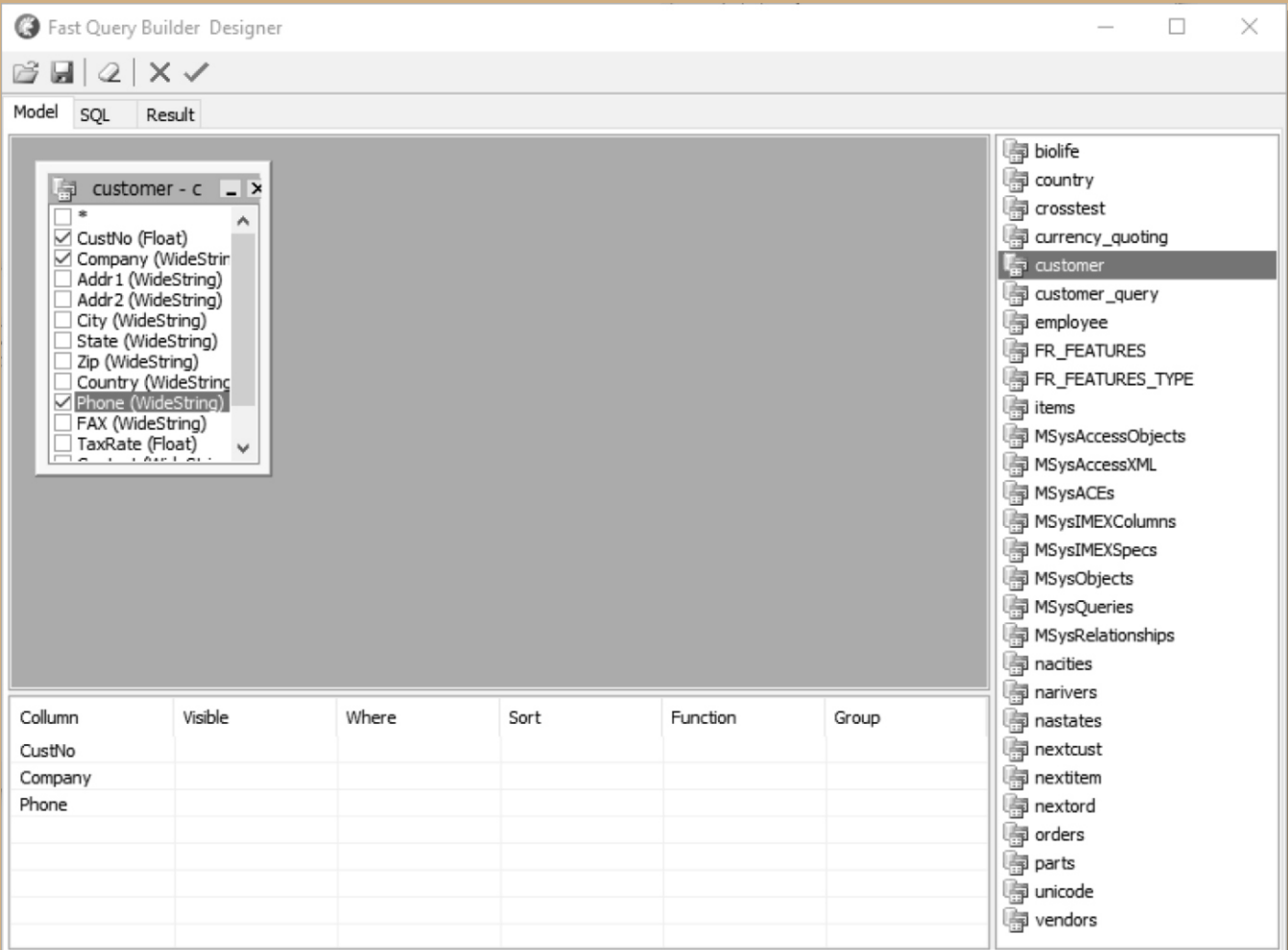
Figure 39: Visual design of the query

This is all that is needed to build a query.
You can view the text of a query on the SQL
tab, and on the Result tab you can see the data
that the query returned. Click the button ☑
to close the constructor. In doing so, we will
return to the window of the query editor,
which now displays the generated query text:

Warning! If you fix the text of the query, you
will lose the scheme (placement of tables in
the query constructor and links between
them). If you do not modify the query text
manually, you can always go to the query
constructor and correct the scheme visually.
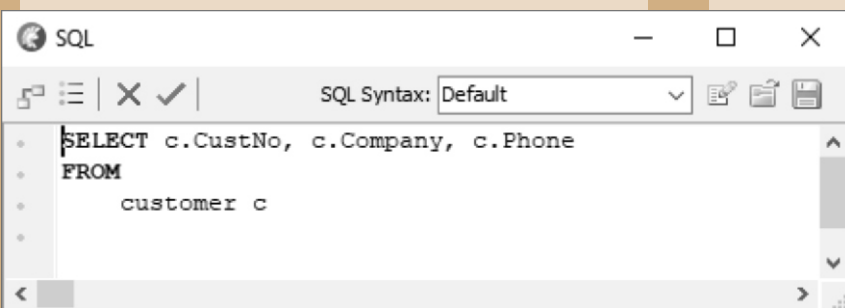
By clicking OK in the editor, we
will return to the report designer.
All we have to do is connect the
"Level 1 Data" band to the data
source and place the fields on
the band.


```
SELECT c.CustNo, c.Company, c.Phone
FROM
    customer c
```

Figure 40: simple SQL

BUILDING A COMPLEX QUERY

In the previous example, we built a report based on data from a single table. Consider building a query that includes data from two tables.

We need to make a query in SQL language, which will return data from both tables, grouped according to a certain condition. In our case, the condition is that the CustNo fields in both tables match.

Like in the previous example, we create a new report and place the "ADO request" component on the page. In the query editor click the button to start the query constructor.

Drag and drop two tables on the working field - Customers and Orders. Both tables have the CustNo field, by which we should link them. By dragging and dropping the CustNo field from one table to another, we create a link between the tables (See Figure xx below):

Now you need to select the fields that should include the query and group it by the CustNo field. To do so, tick the "*" field in both tables and the CustNo field in the Customer table. The fields we have selected will appear at the bottom of the window, after which you should select the sorting for the CustNo field:
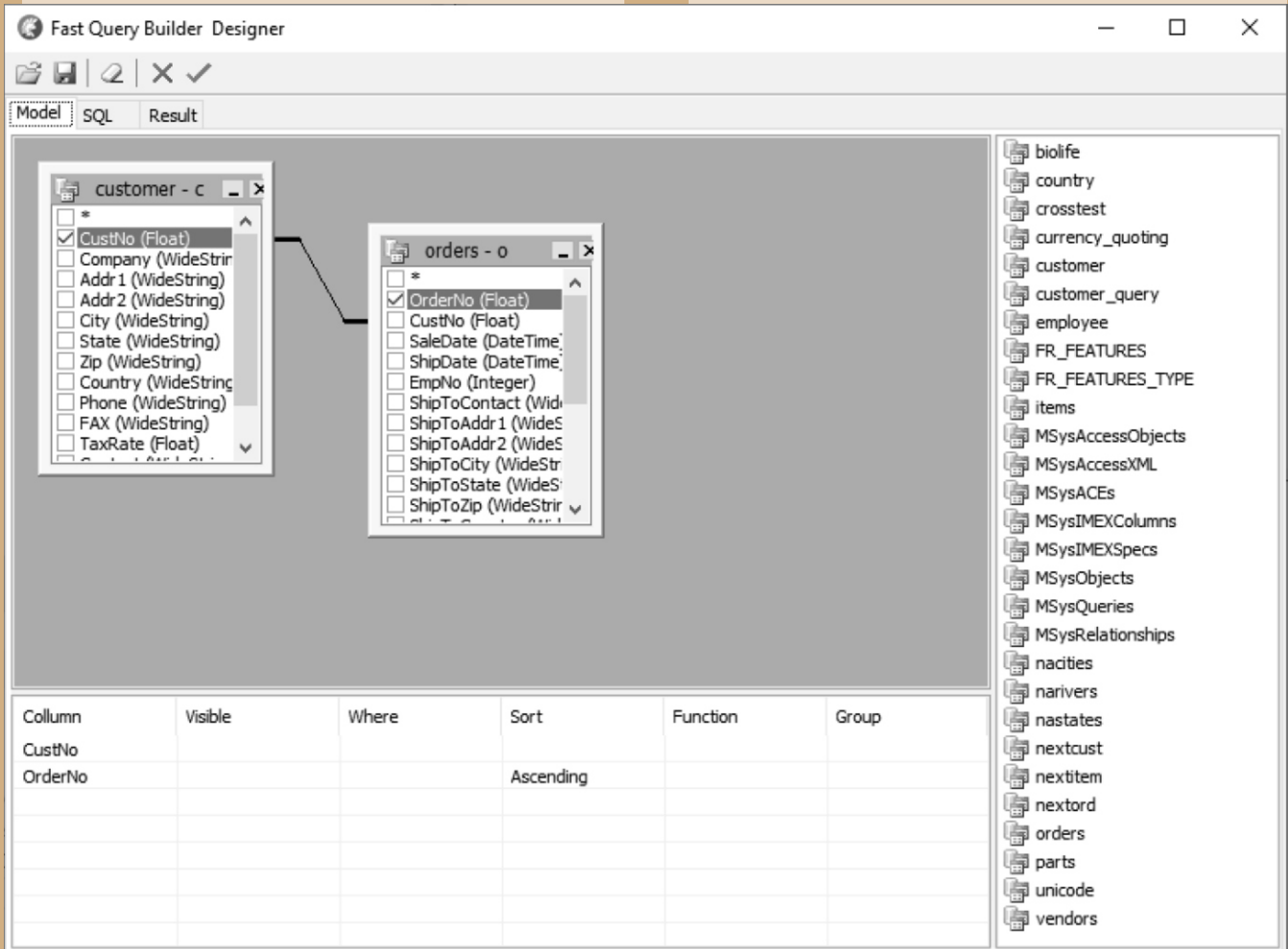
Figure 41: Designing a query with two tables

Collumn	Visible	Where	Sort	Function	Group
CustNo					
OrderNo	⌄		Ascending ⌄	⌄	⌄

Figure 42: Editing the table parameters

All right, the SQL is Finalized now.
Its text looks like this:

```
SELECT c.CustNo, o.OrderNo
FROM
    (customer c
    INNER JOIN orders o ON (c.CustNo=o.CustNo))
ORDER BY o.OrderNo
```

Figure 43: - Request to database

By Detlef Overbeek, code used from Jitendra Gouda

starter ◀ ▶ expert
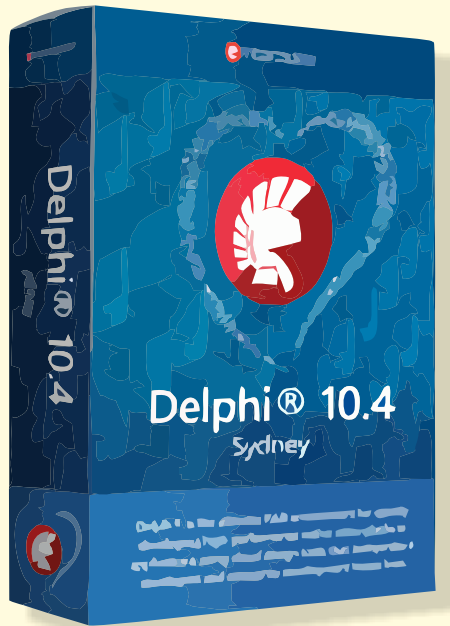
INTRODUCTION:
In the last issue (93) I have shown how to print in Lazarus. As promised here comes a Delphi version. I searched of course for examples. And I found one that almost met my requirements: `http://delphiprogrammingdiary.blogspot.com/2019/03/customized-printing-in-delphi.html` . The author is Jitendra Gouda.

I reconstructed his code to applications and of course added some parts extra. If you study this, you will understand how the printing mechanism works. To make it more interesting I added a complete Library of Glyphs. As a subscriber you can obtain that from your downloadpage `https://www.blaisepascalmagazine.eu/your-downloads/`. (First log in).

With this show-model you can print text, you can use a RichEditMemo and a ListView to make a report. It's not easy but very helpful to understand how to print.

In a separate article I'll show a RichEdit from TMS that can print in a way you never dreamed of, but that's commercial. I here pay extra attention towards built in components in Delphi. Except for the Glyph Library nothing special. And they are for free, like the editor for creating and/or altering them: XNResources. This was explained and shown some issues ago: Nr. 71 and even before (29)- unless you would go back to Delphi 7,which includes the "Image Editor".

I'll again describe what you need to do to make your BitButton or SpeedButton work with small images.
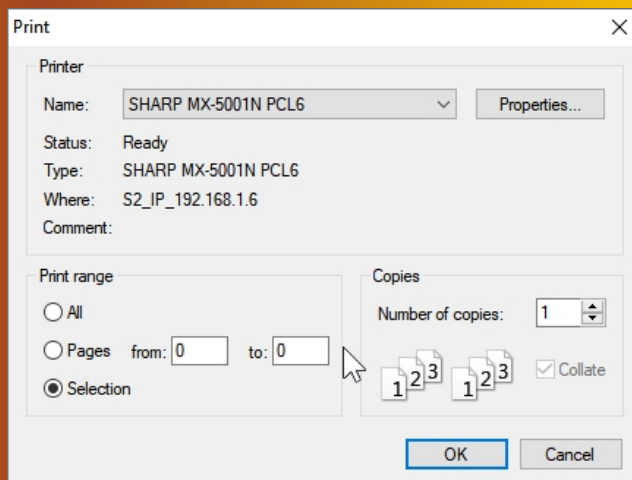
Figure 2: Printing with selection, page range ability and Printerselection - some extra settings must be made.
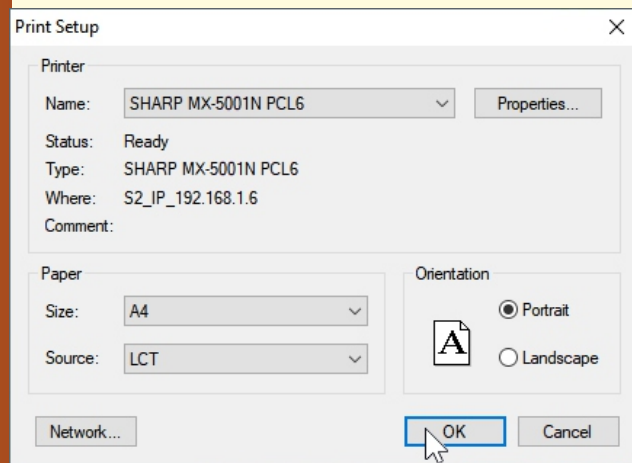
Figure 3: easy overview of what you can achieve, choose from a list of printers, set the size and orientation.
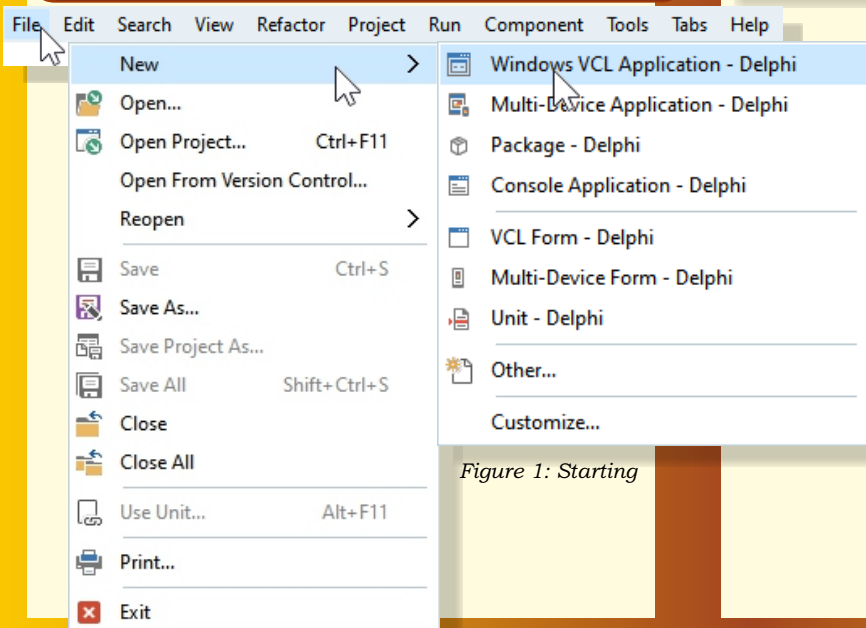
Figure 1: Starting

In this article is shown:
The use of
- **PrinterDialogs**
  (See Above Figure 2 and 3)
  – and special settings
- **OpenTextFile-** and
- **OpenPictureDialog** in combination
  (See Above Figure 2 and 3)
- **RichEditMemo**
(See Above Figure 2 and 3)
  – coloring the text
- Memo adding text from file
- Listview – preparing for use of columns, adding text

Figure 4: The optional settings for enabling the printer items

MaxPage should not be set if you want to be able to print the complete document

poPageNums needs to be set if you want to set the page numbers to be printed

poSelection needs to be set if you want to be able to print a selection

These are items that need to be set: you can not alter them during runtime. They appear but are not availble

Properties of the specific printer

The "PrintRange" is set to "All"

For printing the same document more then once

Pages from Fill out the beginning and ending page

Selection means you can choose which specific page you want to be printed

Figure 5: The available possibilities
- if not pre-set are not possible to enable during runtime

Figure 6: The list of specific printers

Figure 7: The needed components

```
unit U_Print_GUI;
{ Credits:
  this is a remake of the examples made by Jitendra Gouda
  http://delphiprogrammingdiary.blogspot.com/2019/03/customized-printing-in-delphi.html
  His Blog : http://delphiprogrammingdiary.blogspot.com/


interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls, Vcl.Buttons, Vcl.ComCtrls, Printers,
  Vcl.ExtDlgs, Vcl.ExtCtrls, System.ImageList, Vcl.ImgList,
  Vcl.BaseImageCollection, Vcl.ImageCollection;
```

Figure 8: Add Printers to the use clause

STARTING

We begin a new VCL application and now need to add some components: The `Printdialog`, `Printersetupdialog`, `OpenTxtFileDialog` and open `PictureDialog`. (see figure 4). You need to add the `Printers Unit` to the `uses-clause`, otherwise you will get error messages - (See Figure 5).
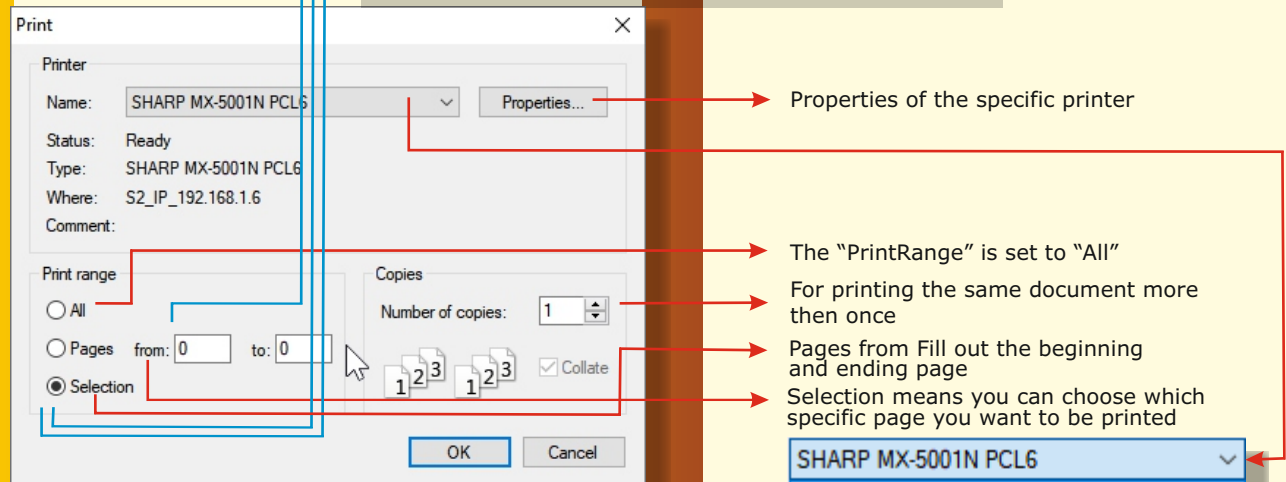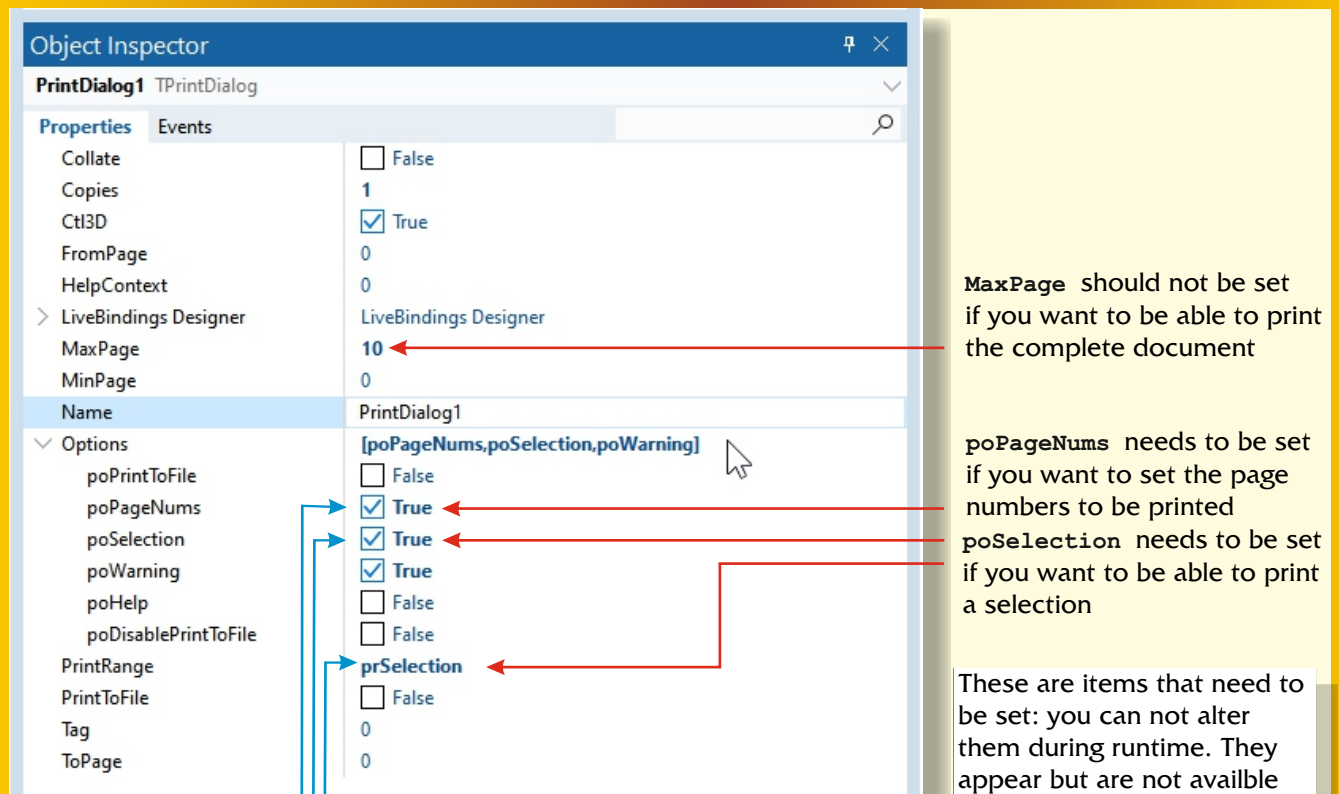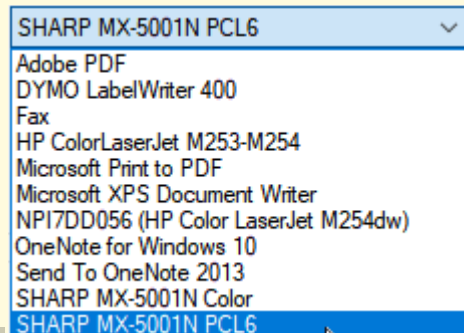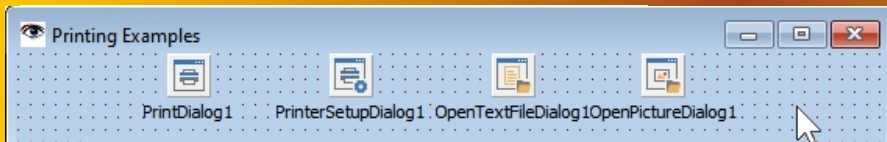
The complete code of this project is also available in your downloads list. Some parts are shown here so you can get an impression of what it looks like. On the first page of this article there are shown two direct elements of the printing topic: If you right click on the dropped `PrintDialog` -component you will be able to select: `TestDialog`.

Imediately starts a wizard as shown on page 1 of this article figure 2.

Now in the **Object Inspector** you can see the settings corresponding to the settings for printing you want to use or allow in your application.

You best test and switch to find out that some settings are interconnected.

Some selections will even not only change, but also exclude others. If the print range is "All" you exclude others: "Pages from" or "Selection". To fully understand you absolutely should change these settings and try them – the only way to find out what you need or want and finally can implement in the application you have or are building.

```
procedure TfrmPrint.FormCreate(Sender: TObject);
begin
  RichEdit1.Clear;
  RichEdit1.SelAttributes.Color := clBlue;
  RichEdit1.SelAttributes.Style := [fsBold];
  RichEdit1.SelText := 'This is bold blue text.';
  RichEdit1.SelAttributes.Color := clRed;
  RichEdit1.SelAttributes.Style := [fsItalic];
  RichEdit1.SelText := #32'This is italic red text';
end;

procedure TfrmPrint.BitBtnPrintMemoClick(
Sender: TObject);
var
  rectPage: Trect; sText: string;
begin
  if PrintDialog1.Execute then
  begin
   rectPage := Rect(
     0, 0, Printer.PageWidth, Printer.PageHeight);
   with Printer do
   begin
    BeginDoc;

    sText := memo1.Text;
    //setting font
    Canvas.Font.Name := 'Verdana';
    Canvas.Font.Size := 11;
    Canvas.Font.Color := clBlack;
    //draws the text with wordbreak
    Canvas.TextRect(rectPage, sText,
       [tfWordBreak]);
    EndDoc;
   end;

  ShowMessage('Printing has been finished.');
 end;
```

Figure 9: Add Printing the RichEditMemo

👁 Printing Examples   —   ☐   ✕

PRINTING EXAMPLE FOR DELPH

| 🖨 Printer Setup | 📇 Print Rich Edit |

Button that prints to the RichEditMemo that is colorized (see Figure 13)

| 🖨 Simple Print Range | 🖨 Print a Memo |

This button allows you to print this "normal" Memo (see Figure 14).

| ✚ Adding Text to Memo |

Here you can add text to the memo by using a text file, searching for the file (see Figure 14).

This is bold blue text. *This is italic red text*

C:
CD "C:\Users\Public\Documents
\Embarcadero\Studio
\21.0\CatalogRepository\AndroidSDK-2525
-21.0.40680.4203\platform-tools\"

An example of what sort of text you can use (see Figure 15).

This is a Rich Edit Memo (see Figure 13).

| ☐ Printing Lines and Shapes | 🖨 Print Image |

ID	Name		CityName	Country
01	Mattias Gaertner		Cologne	Germany
02	Michael van Canneyt		Leuven	Belgium
03	Martin Friebe		Mainz	Germany
04	Detlef Overbeek		IJsselstein	Netherlands
05	Rik Smit		Rotterdam	Netherlands
06	Emiel Overbeek		IJsselstein	Netherlands
07	Vincent Overbeek		Eindhoven	Netherlands
08	Willy Overbeek		IJsselstein	Netherlands
09	Siegfried Zuhr		Boskoop	Netherlands
10	Danny Wind		Rotterdam	Netherlands

| Load Names | Load Citys |

if you use this button it will load the prepared cities and countries

| 🖨 Print the Listview   Report |

Load Names will enter the prepared text into the Listview (see Figure 12 page 6

This button creates the final report (see Figure 12 page 6)

ADDING TEXT

The dialogues are described by their functionality and are shown in the code of the project.

`🖶 Print Rich Edit`

If you want to set the `RichMemo` text you could add this in the form create-procedure (illustrated in the app-overview on page 4 of the article. Please note the overlapping procedures are meant to explain certain ways of handling, they are only examples. So there are so many buttons **(procedures)** in the app simply having different sorts of coding for their events.

A worth knowing part is how to add text to the `Listview`; three buttons help: "**Load names**" (page 6), "**Load City's** and "**Print the ListView**" report. (see the code on page 8). Of course you can change the algorithm to a much shorter and better list like putting it all in arrays etc. but that is out of scope.

In the list view component you can right-click and then the extra's will pop up: choose the **Columns Editor** (see below).

You than can add or remove Columns in the "Structure" - overview you find the example of the editor as well the settings you can make or alter. If you have a "Name" file give it a length value otherwise you might not see it.
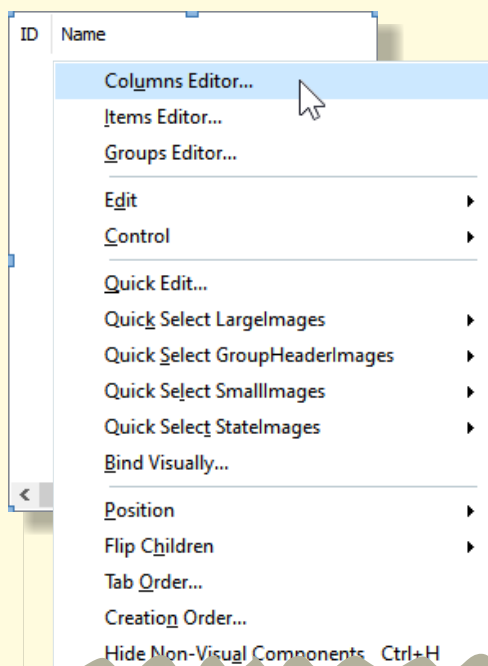
Figure 10: the Columns Editor is used by the preparing for printig a report
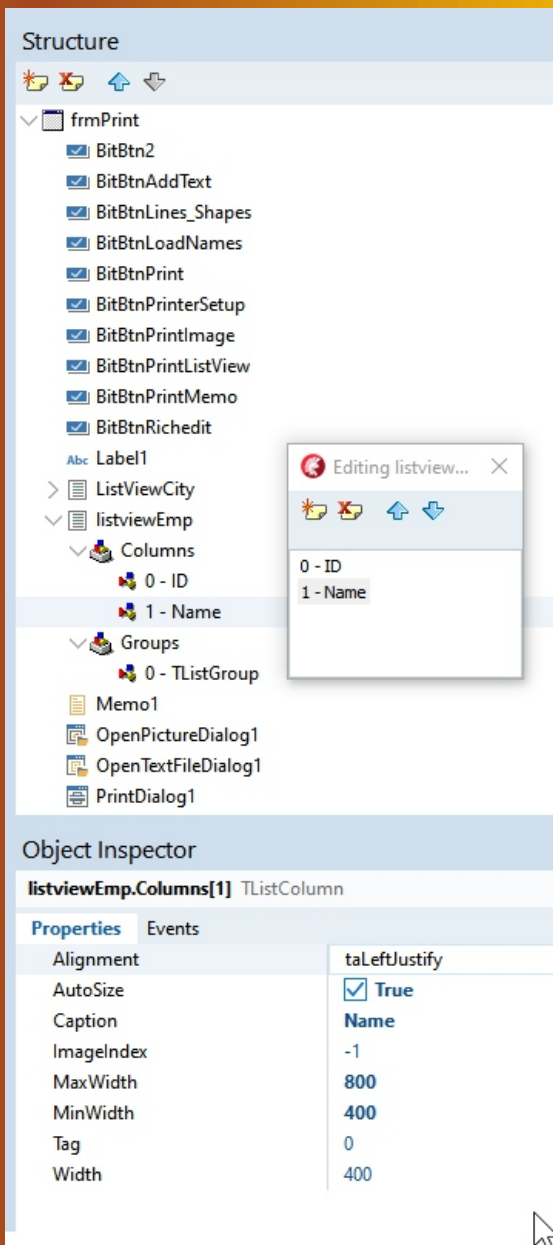
Figure 11: The details are shown...

```pascal
procedure TfrmPrint.BitBtnLoadNamesClick(Sender: TObject);
begin
 with listviewEmp do
 begin
  with Items.Add do
  begin
   Caption := '01';
   SubItems.Add('Mattias Gaertner');
  end;
  with Items.Add do
  begin
   Caption := '02';
   SubItems.Add('Michael van Canneyt');
  end;
  with Items.Add do
  begin
   Caption := '03';
   SubItems.Add('Martin Friebe');
  end;
  with Items.Add do
  begin
   Caption := '04';
   SubItems.Add('Detlef Overbeek');
  end;
  with Items.Add do
  begin
   Caption := '05';
   SubItems.Add('Rik Smit');
  end;

....

  with Items.Add do
  begin
   Caption := '09';
   SubItems.Add('Siegfried Zuhr');
  end;
  with Items.Add do
  begin
   Caption := '10';
   SubItems.Add('Danny Wind');
  end;
 end;
end;
```

Figure 12: Adding names to the ListView

Load Names

Load Citys

Print the Listvieww Report

```pascal
procedure TfrmPrint.BitBtnRicheditClick(Sender: TObject);
begin
 if PrintDialog1.Execute then
 begin
  RichEdit1.Print('Print Rich Edit Data');

  ShowMessage('Printing the RichEdit is done.');

 end;
end;

procedure TfrmPrint.BitBtnPrintImageClick(Sender: TObject);
var
 rectPage: TRect;
 image1: TBitmap;
begin
 if PrintDialog1.Execute then
 begin
  rectPage := Rect(0, 0, Printer.PageWidth, Printer.PageHeight);
  with Printer do

  begin
   if OpenPictureDialog1.Execute then
   //start printing
   BeginDoc;

   image1 := TBitmap.Create;
   image1.LoadFromFile(OpenPictureDialog1.FileName);
   Canvas.StretchDraw(Rect(500, 1000, 1500, 2000), image1);
   image1.Free;
   //finish printing
   EndDoc;
  end;
  ShowMessage('Printing the image is done.');
 end;
end;
```

Figure 13: Print the RichEditMemo

```pascal
procedure TfrmPrint.BitBtnAddTextClick(Sender: TObject);
var
 Encoding : TEncoding;
 EncIndex : Integer;
 Filename : String;
begin
 if OpenTextFileDialog1.Execute(Self.Handle) then
 begin
  //Selecting the file name and encoding
  Filename := OpenTextFileDialog1.FileName;

  EncIndex := OpenTextFileDialog1.EncodingIndex;
  Encoding :=
    OpenTextFileDialog1.Encodings.Objects[EncIndex] as
      Tencoding;

  //Checking if the file exists
  if FileExists(Filename) then
   //Display the contents in a memo based on the selected
encoding.
   Memo1.Lines.LoadFromFile(FileName, Encoding)
  else
   raise Exception.Create('File does not exist.');
  end;
end;
```

Figure 14: The settings for adding text to the memo

```pascal
procedure TfrmPrint.BitBtnPrintClick(Sender: TObject);
begin
 if PrintDialog1.Execute then
 begin
  ShowMessage('Setup Printing is done.');
 end;
end;

procedure TfrmPrint.BitBtnPrinterSetupClick(Sender: TObject);
begin
 if PrinterSetupDialog1.Execute then
 begin
  ShowMessage('The Printer SetUp is done.');
 end;
end;

procedure TfrmPrint.BitBtnLines_ShapesClick(Sender: TObject);
var
 rectPage: TRect;
begin
 if PrintDialog1.Execute then
 begin
  rectPage := Rect(0, 0, Printer.PageWidth, Printer.PageHeight);
  with Printer do
  begin
   //start printing
   BeginDoc;
   Canvas.Pen.Width := 3;
   Canvas.Pen.Color := clBlue;
   Canvas.MoveTo(500, 2200);
   Canvas.LineTo(rectPage.Width-500, 2200);

   Canvas.Brush.Color := clRed;
   Canvas.Rectangle(1000, 2500, 2000, 3500);

   Canvas.Brush.Color := clYellow;
   Canvas.Ellipse(1000, 3000, 2000, 4000);

   //finish printing
   EndDoc;
  end;
 end;
end;

//fill staff + authors member data//
```
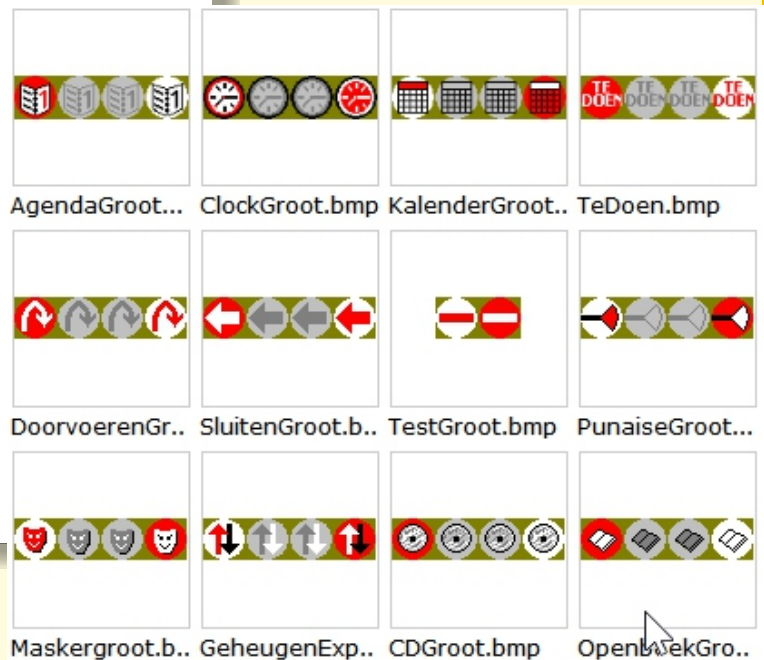

Figure 16: An impression of the library of Glyphs
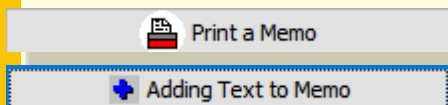
Figure 15: These Buttons print the text as you can see on page

Hint2.bmp	Masker.bmp	ZoekenInText...	Openboek.bmp	PrinterNw.bmp	Lettertype.b..
Tekst.bmp	Info.bmp	About.bmp	Terug2.bmp	Terug1.bmp	Terug.bmp
print3.bmp	Next.bmp	Prior.bmp	Last.bmp	UitWin.bmp	Uitdelph.bmp
ZoomToFit.bmp	zoomback.bmp	Tedoen1.bmp	tedoen.BMP	Minimize.bmp	vinkblw.bmp
Hint1.bmp	voriggr.bmp	Volggr.bmp	Taurus1.BMP	Poort1.bmp	HelpInt1.bmp
Help1.bmp	Help.bmp	UItGroen.bmp	UItBlauw.bmp	LampjeGroen..	LampjeRood1...
LampjeRood.b..	LampjeGroen...	LampjeGeel.b..	LampjeBlauw...	LampjeAan.b..	ptrsetup.bmp
ZoomToWidth...	vorig.bmp	Preview.bmp	Print.bmp	shapes1.bmp	Sysdata.br

There is more if you download them from your personal Download Page

"We got him a phone, computer and his own social media page. That should keep him distracted for a long time."

THE DELPHI COMPANY

-est 1998-

OS X   Android   iOs   Windows

Four Platforms
One develop environment
One Expertise

Vier platforms
Eén ontwikkelomgeving
Eén expertise

DELPHI

www.delphicompany.nl
info@delphicompany.nl

By Danny Wind

starter ———— expert

This series of articles is about writing your own web services server and client in **Delphi.** The approach of all articles is pragmatic.

The first article introduced some of the concepts you need to know and shows you how to create and consume your own web service in **Delphi** with just the `GET` request.

The second article showed you how to update the data in the web service and how to create in-memory storage for the web service.

This third article shows you how to consume and use your web service from both **Delphi** clients on **Windows** and from a web page with **JavaScript.**
It also adds error handling and tweaks some code on the web service which were left as teasers in the previous article.

The web service from the previous article is a functional web service that uses the `HTTP` commands `GET`, `POST`, `PUT` and `DELETE` to get, update, insert or delete items in a key value store.
The key value store holds string keys and string values, and can be used to store **JSON** or other string based data.
The **REST** endpoint we defined was
`http://localhost:8080/KeyValue`
and we can `GET` or `DELETE` a value for a given key using parameters in the URL segment.
`http://localhost:8080/KeyValue/0`
Similarly we can `POST` (update existing) or `PUT` (insert or replace) data
`http://localhost:8080/KeyValue/1/One`
but remember that you need to send a `POST` or `PUT` `HTTP` command, which you can do with the **REST** debugger.
Just opening the above link in a browser would send a `GET HTTP` command.
The `PUT` and `POST` also allow for sending large or complex data within the body of the request instead of using the URL segment.

The Delphi web service client we created in the previous article looked like this
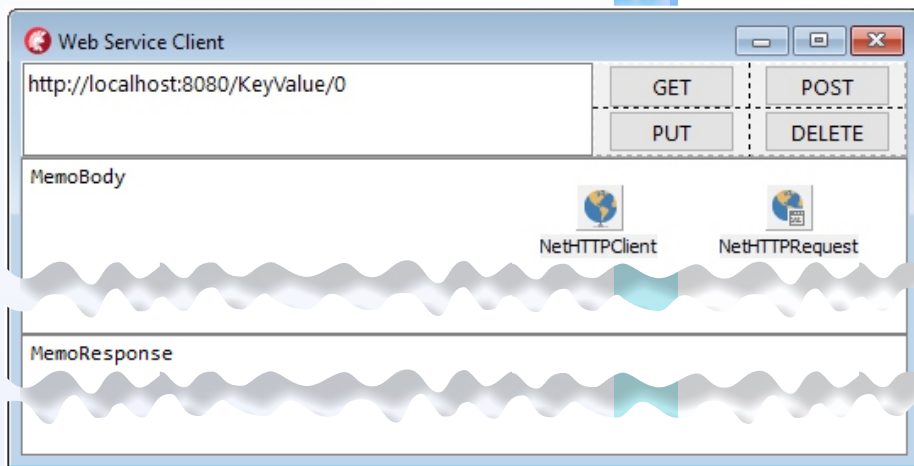
WEB SERVICE PART 3 - CLIENT

By Danny Wind

starter   expert

THE DELPHI COMPANY
-est 1998-

Page 2/14
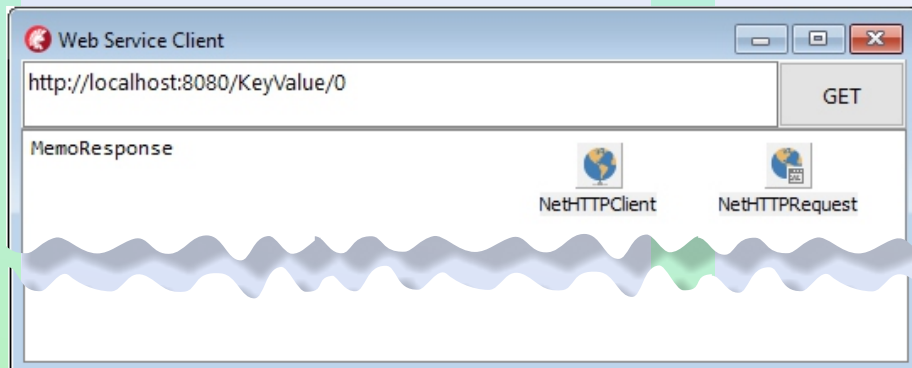
and we will use this web service client as a starting point for our next steps to create a web service client with `POST`, `PUT` and `DELETE` which looks like this:

Open the previous Web Service Client

❶ Add a GridPanel under the **GET button** (you can move components around in the **Structure Viewer**, left in the IDE)

❷ Set the `Align` of the `ButtonGet` to `None` and the `Anchors` to empty

❸ Add three additional Buttons to this **GridPanel** and rename them to `ButtonPost,ButtonPut, ButtonDelete`

❹ Temporarily set `Align` of `MemoResponse` to `None` and move it down

❺ Add a Memo to the Form, place it between the `MemoResponse` and the Edit and then align `Top`

❻ Set `Align` of MemoResponse back to Client

❼ Add an `OnClick` event-handler on the **Button Put** to add the code to **Insert** or **Replace** a value in the web service

```
procedure TFormMain.ButtonPUTClick(Sender: TObject);
var
 lContentStream: TStringStream;
begin
 { Encode string stream as UTF8 }
 lContentStream := TStringStream.Create(MemoBody.Lines.Text, TEncoding.UTF8);
 lContentStream.Seek(0, TSeekOrigin.soBeginning);
 NetHTTPRequest.Put(EditURL.Text, lContentStream, nil, nil);
end;
```

In this code we use the body of the `HTTP` request to send our data with the `NetHTTPRequest.Put`. We could also have added it as a URL segment parameter in the `EditURL.Text`, but that would have more limitations,

in size and in the supported or allowed characters. Because we use a stream, we also need to manually encode the text from the Memo into `UTF-8` which is the default for sending string data to a web service.
This is also more efficient than the Windows default `UTF-16` encoding, resulting in an up to 50% smaller content.

❽ Test if it works by running the web service server from the previous article. You can also use the completed version of the web service server from this article

❾ Use the following `PUT` URL to place a value in key 1 and then use `GET` to retrieve it. The result in the web service client should look like this
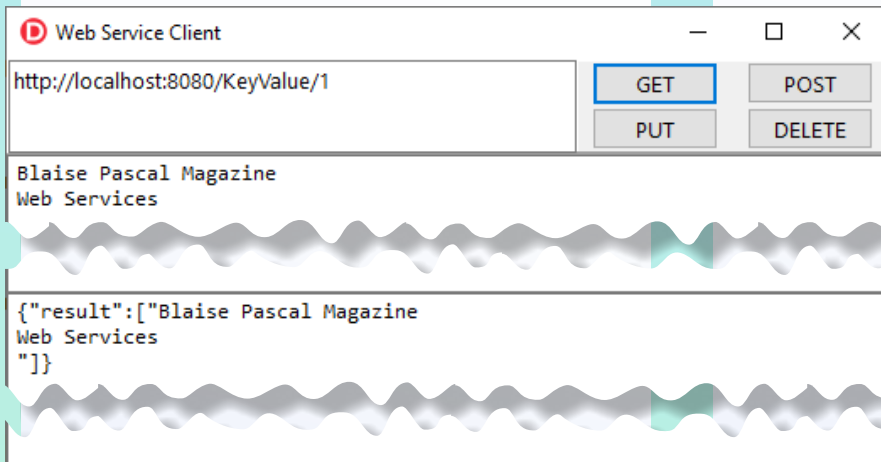
❿ **Notice** how the carriage return - line feeds have also been stored in the `key value` store and they result in multiple lines in the `MemoResponse.` We should encode these special characters to conform to `JSON` standards to prevent other clients from rejecting our malformed `JSON`
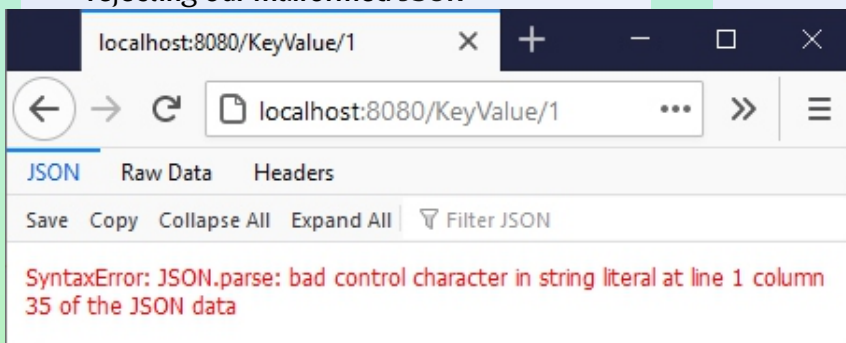
IETF - The Internet Engineering Task Force
specification of the JSON data interchange format
states:
"All Unicode characters may be placed within the
quotation marks, except for the characters that MUST
be escaped: quotation mark, reverse solidus, and the
control characters (U+0000 through U+001F)."
Bray, T., Ed., "The JavaScript Object Notation (JSON)
Data Interchange Format", STD 90,
RFC 8259, DOI 10.17487/RFC8259, December 2017,
`<https://www.rfc-editor.org/info/rfc8259>`.

⓫ To correctly store a string as a JSON string
we need to add JSON string conversion for
the control characters and enclose it in
quotation marks. We use the function
`TJSONString.ToJSON(Options:`
`TJSONOutputOptions)` to convert the
string into a JSON string. Modify the code
as follows

```
procedure TFormMain.ButtonPUTClick(Sender: TObject);
var
 lContentStream: TStringStream;
 lJSONString: TJSONString;
begin
{ Encode string stream as UTF8 }
 lJSONString := TJSONString.Create(MemoBody.Lines.Text);
 lContentStream := TStringStream.Create(
  lJSONString.ToJSON([TJSONAncestor.TJSONOutputOption.EncodeBelow32]),
  TEncoding.UTF8);
 lJSONString.Free;
 lContentStream.Seek(0, TSeekOrigin.soBeginning);
 NetHTTPRequest.Put(EditURL.Text, lContentStream, nil, nil);
end;
```

With this `ToJSON` function the control
characters below `U_001F (32)` are encoded,
where some of the special characters such as
carriage return and line feed are changed to `\r`
and `\n`. Note that I choose to use `ToJSON` with
only `EncodeBelow32` specified. I do not want
Unicode characters above 127 to be encoded
to `\uxxxx`, where `xxxx` is the hexadecimal
value of the UTF-16 characters, as that
would increase the length of our content.
Especially since the latest 2017 ietc
specification states that JSON interchange must
support all UTF-8 characters and escaping
normal UTF-8 characters is not necessary.
⓬   We also need to change a bit of code in
the server, als the stored JSON string already
has its own quotation characters.
For the GET method we modify the code that
returns the JSON array and remove the quotes.
We assume that each stored value is valid JSON
on its own.

```
Response.Content := '{"result":[' + lValue + ']}'
{ this was Response.Content := '{"result":["' + lValue + '"]}';}
```

⓭ Also change the test value for key 0

```
gKeyValueStore.AddOrSetValue('0', '"Zero"');
{this was gKeyValueStore.AddOrSetValue('0', 'Zero');}
```

This is not totally foolproof, as it assumes anyone pushing data into the key value store adds valid JSON, but it's good enough for our simple web service.

⓮ If we now test the service by storing the two lines we get this correct result in the browser
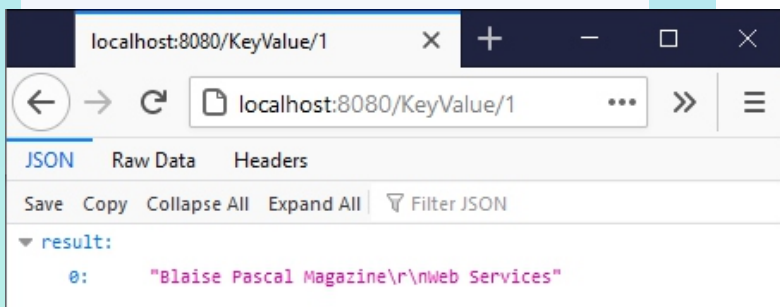
⓯ All looks OK, however if this looks strange to you, remember that we return a JSON array of values with one item (0) with the value for key 1

⓰ Back to the client

⓱ The following code implements the POST functionality

```
procedure TFormMain.ButtonPOSTClick(Sender: TObject);
var
 lContentStream: TStringStream; lJSONString: TJSONString;
begin
 { Encode string stream as UTF8 }
 lJSONString := TJSONString.Create(MemoBody.Lines.Text);
 lContentStream := TStringStream.Create(
  lJSONString.ToJSON([TJSONAncestor.TJSONOutputOption.EncodeBelow32]),
  TEncoding.UTF8);
 lJSONString.Free;
 lContentStream.Seek(0, TSeekOrigin.soBeginning);
 NetHTTPRequest.Post(EditURL.Text, lContentStream, nil, nil);
end;
```

⓲ The code is the same as the PUT, with one additional condition that a POST to a non-existent key will fail with an internal error. Note that the web service server neatly translates such an internal exception to a HTML page

```
Web Service Client                         —    □    ×

http://localhost:8080/KeyValue/1          [ GET ]    [ POST ]
                                          [ PUT ]    [ DELETE ]

MemoBody



<html><body><h1>Internal Application Error</h1>
<p>Item not found
<p><hr width="100%"><i>/KeyValue/1</i></body></html>

```
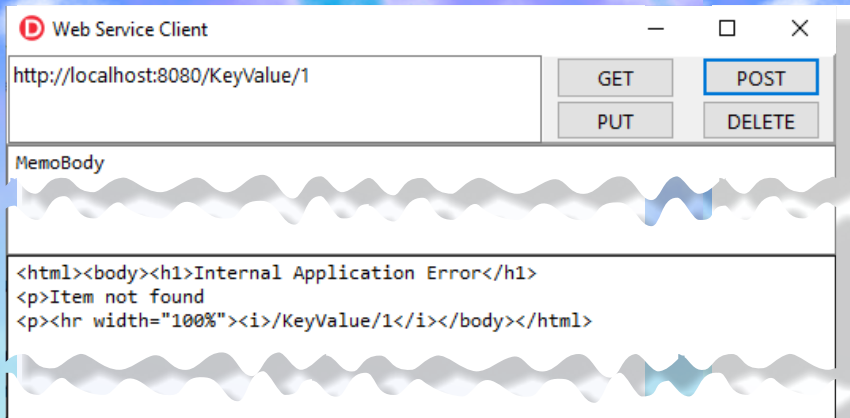
⓳ Instead of this **HTML** page I'd like it to return a **JSON** error

⓴ Open the web service server and find the code that handles the **POST** in the **Web Module** unit

```
{existing code}
procedure TWebModule1.WebModule1WebActionItemKeyValuePOSTAction ...
...
Response.ContentType := 'application/json; charset=UTF-8';
gKeyValueStore[lKey] := lValue;
Response.Content := '{"result":[]}';
Handled := True;
```

㉑ and modify it to return a **JSON** formatted error string if the key is not found in the key value store

```
new code}
procedure TWebModule1.WebModule1WebActionItemKeyValuePOSTAction ...
...
Response.ContentType := 'application/json; charset=UTF-8';
if gKeyValueStore.ContainsKey(lKey) then
begin
 gKeyValueStore[lKey] := lValue;
 Response.Content := '{"result":["OK"]}';
end
else
begin
 Response.Content := '{"error":"Item not found"}';
end;
Handled := True;
```

㉒ and instead of returning an empty **JSON** array, we now also return one array item with "OK", making it easier to parse

㉓ After this code change the result after a click on POST with a non-existent key should look like this

```
Web Service Client                    —    □    ×

http://localhost:8080/KeyValue/1       GET         POST
                                       PUT         DELETE

MemoBody


{"error":"Item not found"}
```
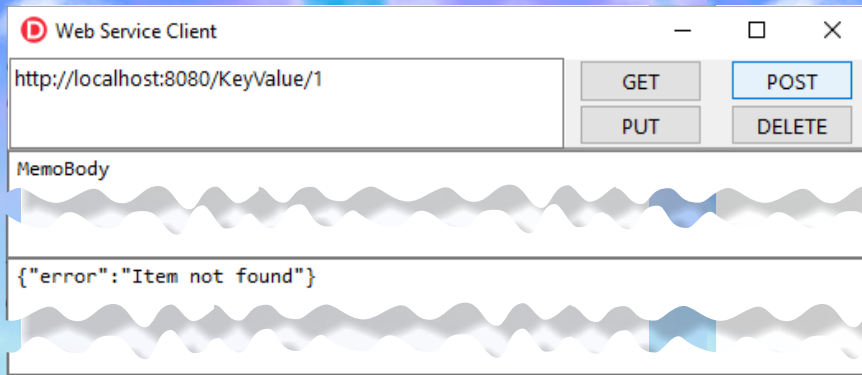
㉔ We go back to the web service client and we finish the client side code with the DELETE

```pascal
procedure TFormMain.ButtonDELETEClick(Sender: TObject);
begin
 NetHTTPRequest.Delete(EditURL.Text, nil, nil);
end;
```

㉕ After which we have a fully functional web services client

The web services client adds values as **JSON** strings, the web services server stores these as-is and when requested returns the **JSON** value as the first item in a **JSON** array.

Maybe at this point you are wondering why we use a **JSON** array to return just one item. That is because using an array is a flexible way of returning items with **JSON.** We can use the **JSON** iterator in a later article to parse for multiple items, for instance if we request the entire list, or if we want to return additional items that describe the content of the value for each key. We could put a value in the key value store that is actually a **BSON** encoded binary file with a descriptor that holds the file type and return the descriptor, which could be a **MIME** type, as an item as well.

On the subject of **MIME** types, there is a small improvement you could make to the header that is sent out by the server. It is currently just a manual string

```pascal
Response.ContentType := 'application/json; charset=UTF-8';
```

but you could change it to

```pascal
Response.ContentType := 'application/json; charset=' + TEncoding.UTF8.MIMEName;
```

This would result in almost the same string, but **UTF-8** would now be written in lower case.

```
'application/json; charset=utf-8'
```

Although using upper case is allowed, as the charset specification is case-insensitive, the default should be in lower case. I just made this mistake when typing the article, as in normal text I tend to use **UTF-8**.
Using **TEncoding.UTF8.MIMEName** instead makes sure I don't repeat that same mistake.

Another thing I forgot to mention was setting the **TNetHTTPRequest** property **Asynchronous** to **True** (default is False) in the web services client. The code also works in synchronous mode, but it is meant to be used asynchronously.

We also have some other things to do that we didn't get around to in the previous article. Let's revisit some code on the server side.

In our previous article we declared and created a global lock variable, but we did not actually use it. If you have Show Error Insight levels set to **"Everything"** under **Tools-Options** in **Delphi** 10.4.2 you'll get a visual indication the code is incomplete if you open the **WebModule** unit

```
initialization

    gLock := TObject.Create;
      H2077 Value assigned to 'gLock' never used  <string, string>.Create;
    gKeyValueStore.AddOrSetValue('0', 'Zero');
240
end.
```

We will add this code soon, but first we dig into the reason why we need to add a global lock.
You may recall that a Web Broker application only has one WebModule class variable as you can see in the interface section of the WebModule unit

```
var
WebModuleClass: TComponentClass = TWebModule1;
```

However for each request a new WebModule instance of this WebModuleClass type may be instantiated and each incoming request is handled in its own thread.

Instantiation of **WebModules** is handled by the `WebRequestHandler`. **WebModule** instances (of the `WebModuleClass`) are kept in a pool in the handler, if one instance is available the `WebRequestHandler` will use that one, if not a new one will be created.

Threading is handled by the Indy `HTTP` Server. By default the `IdHTTPServer` handles each request by creating its own new thread. If we would create the web service server as `ISAPI` or Apache the threading would be handled there.

For us knowing that we have multiple instances of **WebModules** used from multiple threads at the same time, means we will need to serialize access to our one global in-memory `Key Value` store to make it thread safe. This is where we will use the global lock variable `gLock` as a companion lock object for the `TDictionary` in combination with `Tmonitor`.

`TMonitor` is an excellent choice for locking in multi-threaded applications. Internally `TMonitor` first uses spin waits before actually locking, which reduces context switching.

The lock flag is also built into each class in Delphi through the `TObject` base class.

When locking an object it's good practice to use a companion `TObject` instance, instead of just locking the class directly.

This is because for some classes in the **Delphi RTL** `TMonitor` is also used in its internal code. Using `TMonitor` on such a class could lead to deadlocks. Instead just declare a new `TObject` variable, as we do in our code with gLock, to lock access to the key value `TDictionary`.

❷❻ In each of the methods that access the `Key Value` store we add a lock by surrounding it with `TMonitor.Enter` and `Exit.` For the `GET` web action item handler the new code looks like this

```
Response.ContentType :=
 'application/json; charset=' + TEncoding.UTF8.MIMEName;
if TMonitor.Enter(gLock, 500) then
begin
 try
  gKeyValueStore.TryGetValue(lKey, lValue);
 finally
  TMonitor.Exit(gLock);
 end;
end;
if not(lValue.IsEmpty) then
begin
 // {"result":[JSONValue]}
 Response.Content := '{"result":[' + lValue + ']}'
end
else
begin
 // {"error":"Item not found"}
 Response.Content := '{"error":"Item not found"}';
end;
Handled := True;
```

The `TMonitor.Enter` has a timeout parameter, if the lock is not acquired within 500 milliseconds it will return `False` and the `TryGetValue` will not be executed.
Usually the lock will be acquired within < 1 ms, but if the `Key Value` store is busy from multiple threads it may take longer and we do not want to wait indefinitely. Instead getting value will then fail and return a `JSON` error with Item not found. Alternatively you could also handle this with `HTTP` error codes as some web services do.

27 We add similar code for the `DELETE` web action item handler.

```
Response.ContentType := 'application/json; charset=' + TEncoding.UTF8.MIMEName;
if TMonitor.Enter(gLock, 500) then
begin
  try
    gKeyValueStore.Remove(lKey);
  finally
    TMonitor.Exit(gLock);
  end;
end;
```

28 and the `PUT` web action handler

```
Response.ContentType :=
 'application/json; charset=' + TEncoding.UTF8.MIMEName;
if TMonitor.Enter(gLock, 500) then
begin
  try
    gKeyValueStore.AddOrSetValue(lKey, lValue);
  finally
    TMonitor.Exit(gLock);
  end;
end;
```

㉙   and the POST web action item handler

```
Response.ContentType :=
 'application/json; charset=' + TEncoding.UTF8.MIMEName;
if TMonitor.Enter(gLock, 500) then
begin
 try
  if gKeyValueStore.ContainsKey(lKey) then
  begin
   gKeyValueStore[lKey] := lValue;
   Response.Content := '{"result":["OK"]}';
  end
  else
  begin
   Response.Content := '{"error":"Item not found"}';
  end;
 finally
  TMonitor.Exit(gLock);
 end;
end;
Handled := True;
```

㉚   After which we have a fully functional web services server

This web services server does have some limitations. Because it is using a globally locked key value store its performance will suffer as we get more simultaneous users. If they mostly just `GET` data the penalty for global locking is low as getting data out of a dictionary based key value store is a O(1) operation. It is very quick. However inserting `(PUT)` or deleting `(DELETE)` data from the key value store is somewhat slow as it needs to (re)calculate hash values. If you have many concurrent users that also write a lot I would not use this setup, but instead just use a fast database backend. Using a database backend has the added benefit of persistence.
The current key value store holds values in memory, after a reset of the web service the data is gone. For simple web services that need this type of transient storage this approach works fine.

It's time to have some fun with our web services server. Let's add some `JavaScript` to the mix.

In a previous article I wrote that a web service is not that much different from serving web pages from a web server.
In fact you can add web page producers to the web service server we just wrote and have it return a `HTML` page.
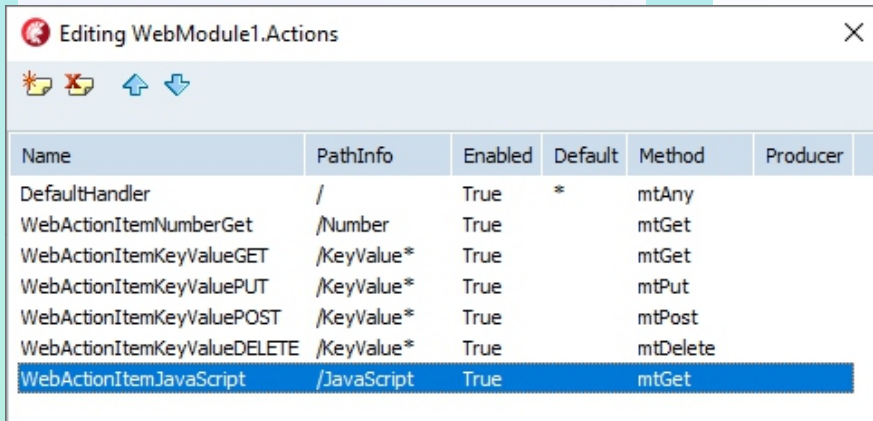We have already seen that when it returned an internal exception as a `HTML` page.

The default handler in the **Web Module** unit does the same thing,it just returns some `HTML`.

This means that we could add an URL to the web service server that would result in a webpage with some `HTML` and a piece of **JavaScript** that would in turn request data from the same web service.

Kind of like a roundtrip, where the web service asks itself a question. This way we would let the web services server serve a web page that acts like a **JavaScript** client to the same web service.

❸❶   We add a new **WebActionItem** handler to the **WebModule** unit, use the **URL /JavaScript** and the method `mtGet`

Editing WebModule1.Actions   ✕

Name	PathInfo	Enabled	Default	Method	Producer
DefaultHandler	/	True	*	mtAny	
WebActionItemNumberGet	/Number	True		mtGet	
WebActionItemKeyValueGET	/KeyValue*	True		mtGet	
WebActionItemKeyValuePUT	/KeyValue*	True		mtPut	
WebActionItemKeyValuePOST	/KeyValue*	True		mtPost	
WebActionItemKeyValueDELETE	/KeyValue*	True		mtDelete	
WebActionItemJavaScript	/JavaScript	True		mtGet	

❸❷   In the handler we respond with a piece of `HTML` with **JavaScript** code

```
<html>
<head><title>Call Number with JavaScript</title></head>
<body>Call Number with JavaScript
<button onclick="getNumber()">
Get Number in Console Log (view Ctrl-Shift-I).</button>
<script type="text/javascript">
function getNumber()
{ let url = 'http://localhost:8080/Number';
  fetch(url).then(resp=> resp.json().then(j=>
         console.log('\nNumber: ', j)));
}
</script>
</body>
</html>
```

❸❸   The resulting Delphi code is this

```delphi
procedure TWebModule1.WebModule1WebActionItemJavaScriptAction
 (Sender: TObject;Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);
begin
 Response.ContentType :=
  'text/html; charset=' + TEncoding.UTF8.MIMEName;
 Response.Content :=
  '<html>' +
  '<head><title>Call Number with JavaScript</title></head>' +
  '<body>Call Number with JavaScript ' +
  '<button onclick="getNumber()">Get Number in Console Log (view Ctrl-Shift-I).</button>' +
  '<script type="text/javascript">' +
  'function getNumber() {' +
  ' let url = "http://localhost:8080/Number";' +
  ' fetch(url).then(resp=> resp.json().then(j=> console.log("\nNumber: ", j)));' +
  '}' +
  '</script>' +
  '</body>' +
  '</html>';
end;
```

❸❹   For web debugging I usually use either
        Firefox or **Chrome,** you can start the web
        debugging with the key combination
        Ctrl-Shift-I

❸❺   Run the web services server, click the Start
        button, then the Browser button and open
        the JavaScript URL
        `http://localhost:8080/JavaScript`

❸❻   The result after clicking the **JavaScrip**t button
        on the page would look like this
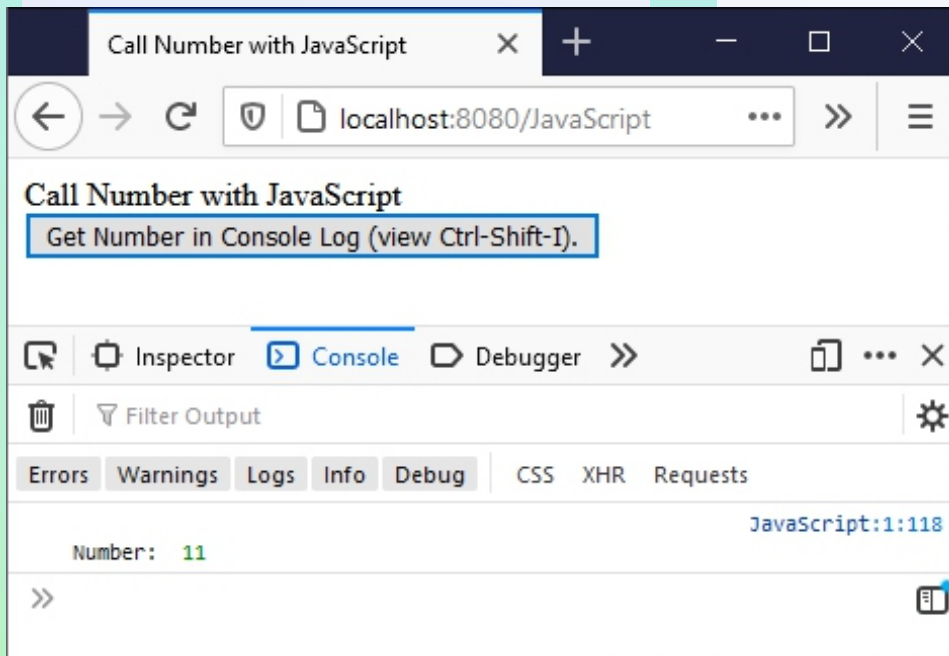
37 The output of clicking the **Get Number** button is only viewable in the Console view in the web debugger (**Ctrl-Shift-I**)

The web service server we made can be used from any other platform that supports web services, from **JavaScript, Python** or **PHP** or any other language or platform.
This makes it a simple solution to enable sharing data from your **Delphi** application with other third-party solutions.

If you start using your **Delphi** web service from other platforms you may run into caching issues and having to configure **Cross-Origin Resource Sharing.**
We will look into these issues in our next article on deployment of the web service server.

A short recap of the things we have done in this article. We created a `POST`, `PUT` and `DELETE` request to the web services client.
We also modified the data to be passed as a **JSON** string adding explicit **UTF-8** encoding of the body content that is sent over the network to the web services server. We added thread-safety code to the web service server and to top it all off we added a `HTML` page with some **JavaScript** code that does a roundtrip and asks our web service for a random number.

In our next articles, we will take a look at **ISAPI** and **Apache** versions of our web services server and also how to deploy each of these to a server and allow access to our web service.
Along the way we will tweak some settings to improve interoperability, with `HTTP` headers for caching and **CORS** and we will configure our web service for better performance. We may also add some more **JSON** support, parsing the result array and adding serialization of objects or even use the web service to store other items besides plain text.
Maybe that last bit will be in another article though. Stay tuned!

PREFACE

In the upcoming **5.16.xx kbmMW Enterprise** Edition will receive another new feature: internationalization also called **I18n**.

The idea about **i18n,** is to be able to translate an application in such a way that it is fully usable in other languages, and preferably it also contains the ability to allow end users easily to switch to their preferred language.

I18N

At first it seems simple… replace all texts with constant codes, and then look those up in a table and output the result of that lookup, and yes… most solutions, including the famous `GetText` `(https://en.wikipedia.org/wiki/Gettext)` solution that many mimics as a defacto.

Basically `GetText` makes a fairly simple lookup to translate one text to a different one.

In later incarnations it has also gained some support for handling translation differently depending on a count.
In many languages a translation which incorporates a count, changes depending on the count number, zero, one or many, and in other cases the count do not matter.

EXAMPLE:
```
I have 1 brother
I have 2 brothers
I have 0 brothers
```

ANOTHER EXAMPLE
```
I have one cookie
I have a few cookies
I have many cookies
I have no cookies
```

In English, it is obvious what the wording should be, but if you would translate the later example to Polish, you would have 4 different texts, while Japanese generally do not change wording based on plurals.

However for kbmMW I have chosen to take it a step further, and have been inspired by a more modern solution that is very popular in modern Javascript based development, **i18next** `(https://www.i18next.com/)`.

What distinguishes **i18next** from `GetText`? **Well one important thing is that i18next supports context sensitive translation.** It is sort of an extension to the plural based translation.

EXAMPLE:
```
My sister she is nice
My brother he is nice
```
In the above example, the context is either sister or brother.
If we are referring to my sister, the sentence will, in English require use of `she` alternatively `he`.
In other languages there may be no differentiations.

ANOTHER EXAMPLE:
One of my secretaries was remarking only this morning how well and young I am looking.
In this example, secretaries is a context word which controls how the translation would be, because it is interpreted as a different gender in different languages.
If another title would be used instead of secretaries, the translation would be different.

```
French: Un de mes secrétaires [male]
Italian: Uno dei miei segretari [male]
Spanish: Una de mis secretarias [female]
Portuguese: Uma das minhas secretárias [female]
German: Einer meiner Sekretäre [male]
```

As you can see, translation is not always just as simple as replacing one text with another, because context may get into the equation.

i18n THE kbmMW WAY
Neither **i18next** nor kbmMW's **i18n** is the perfect tool for translation, but both gets closer to the perfect solution than others who do not consider context sensitive translation.

I looked at **i18next's** configuration files, which defines how translations are done, and figured that they were quite ugly and not really suited for a framework like **Delphi.**

Further i18next only supports one level of context, while I decided to add support for multiple levels of context within the same sentence.

Also **i18next** handles the concept of pluralization as a separate topic, while I found that pluralization is simply a variation of the context.
So by handling multiple levels of context, **kbmMW's i18n** automatically supports both traditional context (for example gender) and pluralization in the same sentence.

Why not just use **Delphi's** built in translation solution? Well the built in **Delphi** translation solution is generally based on constant resource strings, which are compiled to DLLs and used as a simple lookup translation.
So you will need a tool to generate those DLLs and you will not have context sensitive translation.
Further strings in your application will need to take advantage of resource strings and thus define a constant integer value for each string you will want to translate.

It is somewhat cumbersome and IMO not really a good way.

kbmMW's i18n supports loading the internationalization from various storages, which currently includes **JSON** and **YAML** based file formats.
I personally find that **YAML** format is the easiest to digest for the human eye, and it also supports entering comments, which **JSON** does not support.

Other formats can be added, which could cross support existing translation description formats.

To make access to **i18n** easy, a singleton instance is readily available when you include the unit **kbmMWI18N** in your application.
The singleton is named **i18n** and will be used for all translation related functionality.
It is possible to make your own instances of **TkbmMWI18N** if you so wish, but it should rarely be needed.

kbmMW's i18n supports two ways to translate, auto translation of select properties on components and forms/frames, and translation of static and dynamic texts, used within the code. In addition it supports setting other properties, like size and position values, in case components require some slight rearrangement to fit a translation.

TRANSLATION IN CODE
It can be done in a couple of ways, either simple translation of a string without any consideration of context, or translation using a format string, where the arguments can be used as context.

```
ShowMessage(i18n.Translate('This is some text to translate'));
```

It will attempt to translate the static text to the language that is currently selected. We will see shortly how to define languages, how to load them in and how to select them.

```
ShowMessage(i18n.Format('This is %s %d of %d',['brother',1,3]));
```

This is working quite much as a regular **Delphi** format and supports all its format specifiers, but also supports additional formatting of date and time. The arguments will be understood as regular arguments, and optionally (*depending on the language specification*) also as context controlling values.
You can abbreviate Translate and Format by simply using an underscore. The following examples are doing exactly the same as the above examples:

```
ShowMessage(i18n._('This is some text to translate'));
```

and

```
ShowMessage( i18n._('This is %s %d of %d',['brother',1,3]));
```

TRANSLATION OF FORMS AND COMPONENTS

It is simple to translate any component or form.

You just register that component or form with the `i18n` instance. Then the properties of the form/components and sub-components will automatically be attempted to be translated when the current language changes.

The following will be a typical usage, you register the current form self with the **translation framework**, for example in the forms `AfterConstruction` method, or whenever the form has been fully constructed.

```
i18n.RegisterComponent(self);
```

If you have multiple forms, and you instantiate and release them on the fly.

It is good practice to call:

```
i18n.UnregisterComponent(self);
```

before the form/component is released. kbmMW will, however usually detect destruction of a TComponent instance, and deregister it automatically from translation.

LOADING LANGUAGE FILES

As mentioned before, the language file can be in **YAML** or **JSON** format (*or any format that has been registered with* **kbmMW's i18n** *framework*).

A language file can contain a single language translation, or multiple language translations.

Any language is identified by a name.
The name can be anything, but I recommend that it follows the typical `ISO 639-1` language code standard

`(https://www.w3schools.com/tags/ref_language_codes.asp)`

extended with a country code

`(https://www.w3schools.com/tags/ref_country_codes.asp).`

Eg. da-DK, en-US etc.

The following code will load the language(s) defined in the YAML file `translation.yaml`:

```
i18n.Load('','yaml','file:..\\..\\translation.yaml');
```

The first argument to `Load` is the name of the language to load. If an empty string is given, all languages found in the file are loaded.
The next argument is the format to use, and the final argument is the settings for that format. In this case, it refers to a file placed two directories above the executables current run directory.

CHANGING THE CURRENT LANGUAGE

Loading the language(s) do not alter the applications translation. Not until you actively choose to change the current language.

i18n.CurrentLanguage:='da-DK';
If a language with the name da-DK has been loaded, everything will automatically be translated according to that languages translation rules.

Obviously you can query what the currently selected language name is right now, by checking the CurrentLanguage property.

At any time, you can get an array of loaded language names:

```
var
 a:TArray<string>;
begin
 a:=i18n.LanguageNames;
…
end;
```

And you can get language captions and descriptions and, if defined, which graphic files should be used to show their flags:

```
var
  a:TArray<string>;
begin
  a:=i18n.Languages.GetCaptions;
  a:=i18n.Languages.GetDescriptions;
  a:=i18n.Languages.GetFlags(true);
  // Get the file names for the
           small sized flags.
…
end;
```

THE LANGUAGE FILE (YAML VARIANT)
Ok, now I have shown the relevant simple methods and properties, it is time to get into how you describe a language translation file.

To make it simple, I will show an example and explain from the example:

```yaml
languages:
  da-DK:
    caption      : Dansk
    description  : "For folk der bedst forstår Dansk"
    flag         :

      # some small flag
      small: ".\\DK_64x64.png"

      # a larger flag
      large: ".\\DK_512x512.png"

    formatSettings:
      currencyString         : dkr
      currencyFormat         : 3
      currencyDecimals       : 2
      shortDateFormat        : "%D-%M-%Y"
      longDateFormat         : "%D. %M2 %Y"
      shortTimeFormat        : "%H:%N"
      longTimeFormat         : "%H:%N:%S"

      # Short Month Names
      shortMonthNames        : [ Jan, Feb, Mar, Apr, Maj, Jun, Jul, Aug, Sep, Okt, Nov, Dec ]
      longMonthNames         : [ Januar, Februar, Marts, April, Maj, Juni, Juli, August,
                                 September, Oktober, November, December ]
      shortDayNames          : [ Søn, Man, Tir, Ons, Tor, Fre, Lør ]
      longDayNames           : [ Søndag, Mandag, Tirsdag, Onsdag, Torsdag, Fredag, Lørdag ]
      thousandSeparator      : "."
      decimalSeparator       : ","
      twoDigitYearCenturyWindow: 50
      negCurrFormat          : 8
      negativeCurrencyFormat : 1
      dateSeparator          : /
      timeSeparator          : ":"
      listSeparator          : ","
      timeAMString           : AM
      timePMString           : PM

  properties   :
    Form1.btnLoadLanguage.Caption  : Hallo
    Form1.Caption                  : Dansk
    Form1.btnLoadLanguage.Height   : 68

  phrases      :
    Hallo                               : Hallo
    OK                                  : OK
    "Dette er en dato %{SHORTDATE}"     : "Dette er en dato %{SHORTDATE}"
    "Dette er et tidspunkt %{LONGTIME}" : "Dette er et tidspunkt %{LONGTIME}"
    "Dette er en dag %{SHORTDAYNAME}"   : "Dette er en dag %{SHORTDAYNAME}"
    "Dette er en måned %{LONGMONTHNAME}": "Dette er en måned %{LONGMONTHNAME}"
    "Dette er en numerisk værdi %f"     : "Dette er en numerisk værdi %f"
    "Dette er en valuta værdi %c"       : "Dette er en valuta værdi %c"
    "Jeg har %d søster"                 :

      # No CONTEXT definition, so all arguments will be considered context
      "1": "Jeg har 1 søster"
      "*": "Jeg har %d søstre"
```

```
    "Dette er %s %d ud af %d %s"         :
      CONTEXT   : [ 1, 3, 2 ]

      # Propose which placeholders arguments should be considered context defining.
      # Starting with 1. The order of the argument indexes are significant.
      # If CONTEXT not defined, all placeholders arguments will be used in default order.
      # This example provides same result as if CONTEXT was not defined.
      # Arguments are numbered from 1. Syntax %{n:format} allows reordering arguments on
        translation.

      søster/1/1: "This is %{2:%d}. sister of %{3:%d} sisters"
      bror/1/1  : "This is %{2:%d}. brother of %{3:%d} brothers"
      søster/0  : "There are no sisters"
      bror/0    : "There are no brothers"
      søster    : "This is %{2:%d}. sister of %{3:%d} sisters"
      bror      : "This is %{2:%d}. brother of %{3:%d} brothers"
      "*"       : "This is %{2:%d}. %{1:%s} of %{3:%d} %{4:%s}"

    Form1                                : "Dansk"
    "Current language:da-DK"             : "Nuværende sprog:da-DK"
    "Load language"                      : "Indlæs sprog"
    Learning                             : Lær
    "Save language"                      : "Gem sprog"
    Translate                            : Oversæt
    "Simple translate"                   : "Simpel oversættelse"
    "Format translate"                   : "Formateret oversættelse"
    "Memo1\r\n"                          : "Dansk data i Memo1\r\n"
    "Current language:%s"                : "Nuværende sprog:%s"
    "Learn phrases"                      : "Lær sætninger"
    "Learn properties"                   : "Lær properties"

  propertyNames : [ Text, Caption, Hint, Width, Height ]

en-GB:
  caption      : English
  description  : "For people who best understands English"
  flag         :
    small: ".\\UK_64x64.png"
    large: ".\\UK_512x512.png"

  formatSettings:
    currencyString           : "$"
    currencyFormat           : 2
    currencyDecimals         : 2
    shortDateFormat          : "%M/%D/%Y"
    longDateFormat           : "%M2 %D. %Y"
    shortTimeFormat          : "%H:%N"
    longTimeFormat           : "%H:%N:%S"
    shortMonthNames          : [ Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec ]
    longMonthNames           : [ January, February, March, April, May, June, July, August,
                                 September, October, November, December ]
    shortDayNames            : [ Sun, Mon, Tue, Wed, Thu, Fri, Sat ]
    longDayNames             : [ Sunday, Monday, Tuesday, Wednesday, Thursday,
                                 Friday, Saturday ]
    thousandSeparator        : "\0"
    decimalSeparator         : "."
    twoDigitYearCenturyWindow: 50
    negCurrFormat            : 8
    negativeCurrencyFormat   : 1
    dateSeparator            : /
    timeSeparator            : ":"
    listSeparator            : ","
    timeAMString             : AM
    timePMString             : PM
```

```
properties  :
   Form1.btnLoadLanguage.Caption    : Hello
   Form1.Caption                    : English
   Form1.btnLoadLanguage.Height     : 38

 phrases    :

   # Simple translation
   Hallo                            : Hello
   OK                               : OK
   søster                           : sister


   #
   # Translation with numerical values and no context variations.
   "Dette er en numerisk værdi %f"  : "This is a numerical value %f"
   "Dette er en valuta værdi %c"    : "This is a currency value %c"


   #
   # Translation with functional arguments and no context variations.
   "Dette er en dato %{SHORTDATE}"      : "This is a date %{SHORTDATE}"
   "Dette er et tidspunkt %{LONGTIME}"  : "This is a time %{LONGTIME}"
   "Dette er en dag %{SHORTDAYNAME}"    : "This is a day %{SHORTDAYNAME}"
   "Dette er en måned %{LONGMONTHNAME}" : "This is a month %{LONGMONTHNAME}"


   #
   # Translation with count context variations. Only one value is provided
   "Jeg har %d søster"                  :

     # No CONTEXT definition, so all arguments (1) will be considered context
     "1": "I have one sister"
     "*": "I have %d sisters"                        # Fallback translation.


   #
   # Context specific translation.
   "Jeg har 1 %s"                   :
     søster: "I have one sister"
     bror  : "I have one brother"
     "*"   : "I have one unknown affiliate"


   #
   # Context and count specific translation. Context is given as arguments.
   "Dette er %s %d ud af %d %s"      :
     CONTEXT   : [ 1, 3, 2 ]
     # Propose which placeholders arguments should be considered context defining.
     # Starting with 1. The order of the argument indexes are significant.
     # If CONTEXT not defined, all placeholders arguments will be used in default order.
     # Only first 3 arguments of the 4 provided are considered context and in the specific
       order 1,2,3.
     # Optional context arguments must be last.
     # Arguments are numbered from 1. Syntax %{n:format} allows reordering arguments on
       translation.
     søster/1/1: "This is %{2:%d}. sister of %{3:%d} sisters"
     bror/1/1  : "This is %{2:%d}. brother of %{3:%d} brothers"
     søster/0  : "There are no sisters"
     bror/0    : "There are no brothers"
     søster    : "This is %{2:%d}. sister of %{3:%d} sisters"
     bror      : "This is %{2:%d}. brother of %{3:%d} brothers"
     "*"       : "This is %{2:%d}. %{1:%s} of %{3:%d} %{4:%s}"
```

```
     Form1                                 : English
     "Current language:da-DK"             : "Current language:da-DK"
     "Load language"                      : "Load language"
     Learning                             : Learning
     "Save language"                      : "Save language"
     Translate                            : Translate
     "Simple translate"                   : "Simple translate"
     "Format translate"                   : "Format translate"
     "Memo1\r\n"                          : "Memo1\r\n"
     "Current language:%s"                : "Current language:%s"
     "Learn phrases"                      : "Learn phrases"
     "Learn properties"                   : "Learn properties"

   propertyNames : [ Text, Caption, Hint, Width, Height ]
```

The **YAML** file is positional aware.
Hence everything that belongs together must start at the same position on a line.
Further **YAML** property/object names are case sensitive.

In the above example you will notice that an **YAML** described object named languages has been defined.
The object contains a number of properties, which each of them are also objects.
The first one is called `da-DK` and the second one is called `en-GB`.

These objects contain the actual language translation settings for that particular language. You can have one or more unique language objects in each file.

Each language object, have an optional **Caption** and an optional **Description,** and an optional set of flag graphics file paths, one named small and one named large.

Then comes a **formatSettings** object, which in turn contains the settings for **TFormatSettings**. You can lookup the **Delphi** manual for an explanation about its settings.
However there is a significant difference in the sense that **shortDateFormat, longDateFormat, shortTimeFormat** and **longTimeFormat** use **TkbmMWDateTime** format specifiers which you can read more about here:

`https://components4developers.blog/2018/05/25/kbmmw-features-3-datetime/`

Next optional object which is named properties, lists any properties for any instantiated component in your application, for which you want set to a specific value. It can be string or numerical properties, and thus allows you to resize or rearrange various controls if needed to make it perform well with your translation.

The values specified here will be used without further translation.

The optional object called **propertyNames** controls which properties will be scanned for potential translation. In this example, only properties named **Text, Caption, Hint, Width** or **Height** will be potentially translated. If **propertyNames** is empty or not specified, then all non empty string properties will be eligible for translation.

Finally we have the general phrase translation object which should be used for the bulk of translation.
The phrase object will contain any number of unique phrases, as you have defined them in your application. String properties on components, that has not already been translated in the properties section, will be attempted to be translated via the phrases section. The same will all runtime string translations, using **Format, Translate** or the **_ methods.**

A translation can be as simple as stating the original value and the translated value, but when using the Format, the arguments may need to be rearranged to make a correct translation, or perhaps the format is context sensitive. Those features can all be provided in the phrases object.

EXAMPLE OF A SIMPLE TRANSLATION:

```
Hallo    : Hello
```

If the string Hallo is being used with Translate or _ or any property that is not listed in the properties section contains the string Hallo, it will be automatically translated to Hello if the en-GB language is selected.

EXAMPLE OF A SIMPLE FORMAT TRANSLATION:

```
"Dette er en numerisk værdi %f" : "This is a numerical value %f"
```

In this case, we make a simple translation, but includes a format specifier for a floating point value. This is used when calling `Format` or _ in code.

EXAMPLE OF A REARRANGED FORMAT TRANSLATION:

```
"Side %d ud af %d sider" : "Total %{2:%d} pages. This is page %{1:%d}"
```

In this example the order of arguments has changed in the translation.
The index of the first argument in the original string is 1, the next is 2 etc. The full format specifier, in this case %d, can be copied over to the translated variant after the colon.

EXAMPLE OF A SIMPLE CONTEXT SPECIFIC FORMAT TRANSLATION:

```
"Jeg har %d søster":
    "1": "I have one sister"
    "*": "I have %d sisters"
```

This example makes a context specific translation depending on the argument given to `Format` or _. If the argument is 1, then the text will be translated to "I have one sister".
In all other situations, the translation will be "I have n sisters" where n is the actual number given as the argument. Hence "*" is the default translation for the phrase if no other contexts matches.

EXAMPLE OF MULTIPLE ARGUMENT CONTEXT SPECIFIC FORMAT TRANSLATION:

```
"Jeg har %d %s":
    CONTEXT    : [ 2, 1 ]
    "mand/1"   : "I have one man"
    "mand"     : "I have %{1,%d} men"
    "kvinde/1" : "I have one woman"
    "kvinde"   : "I have %{1,%d} women"
    "*"        : "I have %d %ss"
```

This translation is triggered by the Format or _ methods like this:

```
ShowMessage(_('Jeg har %d %s',[1,'mand']));
```

In English, the Danish word 'mand' will be translated to either man or men, depending on the count, and similarly the Danish word 'kvinde' will be translated to either woman or women.

For the English language, we want the 'mand' or 'kvinde' word to be the primary context word. For that reason I have defined a **CONTEXT** property which controls in which order the arguments are used in building the context specifier. It is perfectly legal to omit arguments if they have no relevance in the context specifier. Eg:

```
"Jeg har %d %s":
    CONTEXT : [ 2 ]
    ...
```

Then only the string argument will be used for defining the context specifier.

In this case I want the 'mand'/'kvinde' word to appear first, and the count after. That makes it possible to define wild card style context specifiers, like "mand/1" and "mand", where the first specifier matches specifically the word 'mand' and the count 1, while "mand" matches "mand" with any count. Contexts are attempted matched with most precise context match first. If none are found, another iteration is attempted, without the least significant context word and so on, until either a match has been found, or nothing is matched, after which the "*" match is used.
If no "*" match has been defined, the original string will be used untranslated.

If no CONTEXT property is given, all arguments will be used for defining the context specifier in their original order.

EXAMPLE OF TRANSLATION WITH FUNCTIONAL ARGUMENTS

```
"Dette er en dato %{SHORTDATE}" : "This is a date %{SHORTDATE}"
```

This example shows how to use Format or _ to output a date, which will be autoformatted according to the chosen language.

Currently a number of functional arguments are supported:

```
SHORTDATE – Converts a floating point, TDateTime or a TkbmMWDateTime value to a short date.
LONGDATE – Converts a floating point, TDateTime or a TkbmMWDateTime value to a long date.
SHORTTIME – Converts a floating point, TDateTime or a TkbmMWDateTime value to a short time.
LONGTIME – Converts a floating point, TDateTime or a TkbmMWDateTime value to a long time.
ISO8601 – Converts a floating point, TDateTime or a TkbmMWDateTime value to an ISO8601 date/time.
```

The functional argument can be prepended with a argument index number to pick the relevant argument for translation, if more are provided.

```
"Velkommen %s. Dette er en dato
%{SHORTDATE}"     : "At date
%{2:SHORTDATE}, we welcome %{1:%s}"
```

This is the first version of I18n for kbmMW, and I'm certain new features will be added as it matures, and requirements are detected.
One of the next things to add, is the integration between kbmMW SmartBind and kbmMW I18N.

Happy translating!

Facts About Bumblebees
By Alina Bradford

Bumblebees are large, fuzzy insects with short, stubby wings. They are larger than honeybees, but they don't produce as much honey. However, they are very important pollinators. Without them, food wouldn't grow.
Two-thirds of the world's crop species depend on animals to transfer pollen between male and female flower parts, according to ecologist Rachel Winfree, an assistant professor in the department of entomology at Rutgers University. Many animals are pollinators — including birds, bats and butterflies — but "there's no question that bees are the most important in most ecosystems," she said in a 2009 article in National Wildlife magazine. While other animals pollinate, bumblebees are particularly good at it. Their wings beat 130 times or more per second, according to the National Wildlife Federation, and the beating combined with their large bodies vibrates flowers until they release pollen, which is called buzz pollination. Buzz pollination helps plants produce more fruit.

A rusty patched bumblebee collects pollen and nectar from a flower. (Image credit: Dan Mullen, `"https://www.flickr.com/photos/8583446@N05/"`

SIZE
There are over 255 species of bumblebees, according to the Integrated Taxonomic Information System (ITIS), so bumblebees can be many sizes.
The largest is the queen of the Bombus dahlbomii, which can grow up to 1.6 inches (4 centimeters) long.

This is three to four times longer than the American bumblebee, according to Scientific American.

FLIGHT
It has often been said that bumblebees defy aerodynamics and should not be able to fly. However, a recent study resolved the enigma and showed how the tiny wings keep the bee in the air. The study, published in the journal Proceedings of the National Academy of Sciences in 2005, used high-speed photography to show that bumblebees flap their wings back and forth rather than up and down.

The wing sweeping is a bit like a partial spin of a "somewhat crappy" helicopter propeller, researcher Michael Dickinson, a professor of biology and insect flight expert at the University of Washington, told Live Science in a 2011 article. However, the angle to the wing also creates vortices in the air — like small hurricanes. The eyes of those mini-hurricanes have lower pressure than the surrounding air, so, keeping those eddies of air above its wings helps the bee stay aloft. [Related: Explained: The Physics-Defying Flight of the Bumblebee]

THE PHYSICS-DEFYING FLIGHT OF THE BUMBLEBEE
Bees have surprisingly fast color vision, about 3 to 4 times faster than that of humans depending on how it's measured, a new study finds.

Short and stubby, the bumblebee doesn't look very flight-worthy.
Indeed, in the 1930s, French entomologist August Magnan even noted that the insect's flight is actually impossible, a notion that has stuck in popular consciousness since then.

Now, you don't need to be a scientist to raise an eyebrow at this assertion, but it sure is easier to explain the bumblebee's physics-defying aerodynamics if you're Michael Dickinson, a professor of biology and insect flight expert at the University of Washington.
`https://www.youtube.com/watch?v=W2YEzY8tzMU`

Facts About Bumblebees

"The whole question of how these little wings generate enough force to keep the insect in the air is resolved," Dickinson told Life's Little Mysteries. "There are details remaining, but it's just not an enigma anymore."

Dickinson published a 2005 study in the journal Proceedings of the National Academy of Sciences on the flight of the bumblebee after gathering data using high-speed photography of actual flying bees and force sensors on a larger-than-life robotic bee wing flapping around in mineral oil. He says the big misconception about insect flight and perhaps what tripped up Magnan is the belief that bumblebees flap their wings up and down. "Actually, with rare exceptions, they flap their wings back and forth," Magnan said.

Take your arm and put it out to your side, parallel to the ground with your palm facing down. Now sweep your arm forward. When you reach in front of you, pull your thumb up, so that you flip
your arm over and
your palm is upwards.
Now, with your palm up,
sweep your
arm back.
When you
reach behind
you, flip your
hand
over again,
palm down for the
forward stroke. Repeat.
If you gave your hand a
slight tilt (so that it's not
completely parallel to
the ground), Dickinson
said, you'd be doing
something similar to
a bug flap.

```
https://www.youtube.com/
watch?v=yRE2rMIXvyU
or
{youtube yRE2rMIXvyU}
or
https://www.youtube.com/
watch?v=UZHa5Y8tZm8&t=14s
```

The fluid dynamics behind bumblebees' flight are different from those that allow a plane to fly. An airplane's wing forces air down, which in turn pushes the wing (and the plane it's attached to) upward. For bugs, it isn't so simple. The wing sweeping is a bit like a partial spin of a "somewhat crappy" helicopter propeller, Dickinson said, but the angle to the wing also creates vortices in the airlike small hurricanes. The eyes of those mini-hurricanes have lower pressure than the surrounding air, so, keeping those eddies of air above its wings helps the bee stay aloft.

Other studies have confirmed that bees can flyin one of the more colorful projects, in 2001, a Chinese research team led by Lijang Zeng of Tsinghua University glued small pieces of glass to bees and then tracked reflected light as they flew around in a laser array. But now, Dickinson says, researchers are more interested in the finer points of how insects control themselves once they're in the air. Those studies will be especially important for a fleet of robotic insects in development, including robobees created by a team at Harvard University.

HABITAT

With so many species, it isn't surprising that bumblebees are found all over the world. For example, the largest bumblebee is found in Argentina and Chile and the rusty patched bumblebee is found in the United States and Canada.
Bumblebees usually build their nests close to the ground — under piles of wood, dead leaves and compost piles — or even below ground in abandoned rodent tunnels, according to Orkin.

Facts About Bumblebees

HABITS

Bumblebees are some of the most social creatures in the animal kingdom. A group of bumblebees is called a colony. Colonies can contain between 50 and 500 individuals, according to the National Wildlife Federation. A dominant female called the queen rules the colony. The other bees serve her or gather food or care for developing larvae. During the late fall, the entire colony dies, except for the queen. She hibernates during the winter months underground and starts a new colony in the spring.

DIET

Bumblebees eat nectar and pollen made by flowers. The sugary nectar provides the bees with energy while the pollen provides them with protein, according to The Bumblebee Conservation Trust. They make honey by chewing the pollen and mixing it with their saliva, according to Animal Diversity Web (ADW). They feed the honey to the queen and the developing brood.

OFFSPRING

The queen is the mother of all the bees in a colony. After waking from hibernation, the queen finds food and looks for a good location for a nest. Once the nest is found, she lays her eggs and stores up food for herself and the babies, according to ADW.

The queen sits on the eggs for about two weeks to keep them warm. When the eggs hatch, the queen feeds pollen to the baby bees, called larvae. At two weeks old, the larvae spin cocoons around themselves and stay there until they develop into adult bees. The queen only takes care of the first batch of babies.

The first batch grows into worker bees that will clean and guard the nest, find food and take care of the next batch of baby bees. The queen is left to do nothing but lay and hatch new eggs.
Bees born in late summer are male bees, called drones, and future queen bees. Both leave the nest as soon as they are mature. The males from other nests mate with future queens and then die. After mating, the future queens fatten themselves up and hibernate throughout the winter.

Compared to wasps, bumblebees are quite gentle and docile. They generally are not inclined to sting unless their nests are disturbed, and spend their days buzzing from flower to flower as they collect pollen. They dwell in ground nests and die when autumn rolls around. (Image credit: Ron James)

CLASSIFICATION/TAXONOMY

Here is the taxonomy of bumblebees, according to ITIS:
Kingdom: Animalia Subkingdom: Bilateria Infrakingdom: Protostomia Superphylum: Ecdysozoa Phylum: Arthropoda Subphylum: Hexapoda Class: Insecta Subclass: Pterygota Infraclass: Neoptera Superorder: Holometabola Order: Hymenoptera Suborder: Apocrita Infraorder: Aculeata Superfamily: Apoidea Family: Apidae Subfamily: Apinae Tribe: Bombini Genus: Bombus

Facts About Bumblebees

CONSERVATION STATUS

Many bumblebees are listed as endangered, vulnerable or near threatened by the International Union for Conservation of Nature and Natural Resource's Red List of Threatened Species.

The variable cuckoo bumblebee is listed as critically endangered by the IUCN and is considered one of the rarest species in North American. The rusty patched bumblebee is also listed as critically endangered, and in early 2017 it became the first wild bee in the continental United States to get federal protection under the Endangered Species Act, according to Scientific American.

There is a lot of discussion as to why the overall be population is declining. Some scientists think that there may be a sickness killing off the bees. Others think pollution, global warming or lack of native flowers may be to blame.

OTHER FACTS

Bumblebees are larger than honey bees and generate more heat. This allows them to work during cooler weather.Bumblebees don't die when they sting. This is trait found in honey bees. Bees are covered in an oil that makes them waterproof. Queens shiver to warm up and keep eggs toasty.

https://www.youtube.com/watch?v=RH7BPw4pQTg