//

JSON Automation

maXbox Starter 85 - JSON Automation with Json4Delphi

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

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.

Lets start with a simple sample:

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

First we create an object and parse it:

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

Now in Json4Delphi we can ask the type:

```
   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:

```
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!

```
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:

```
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:

```
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. lets do 2 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: '+ ajt.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'].as
object['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;
```

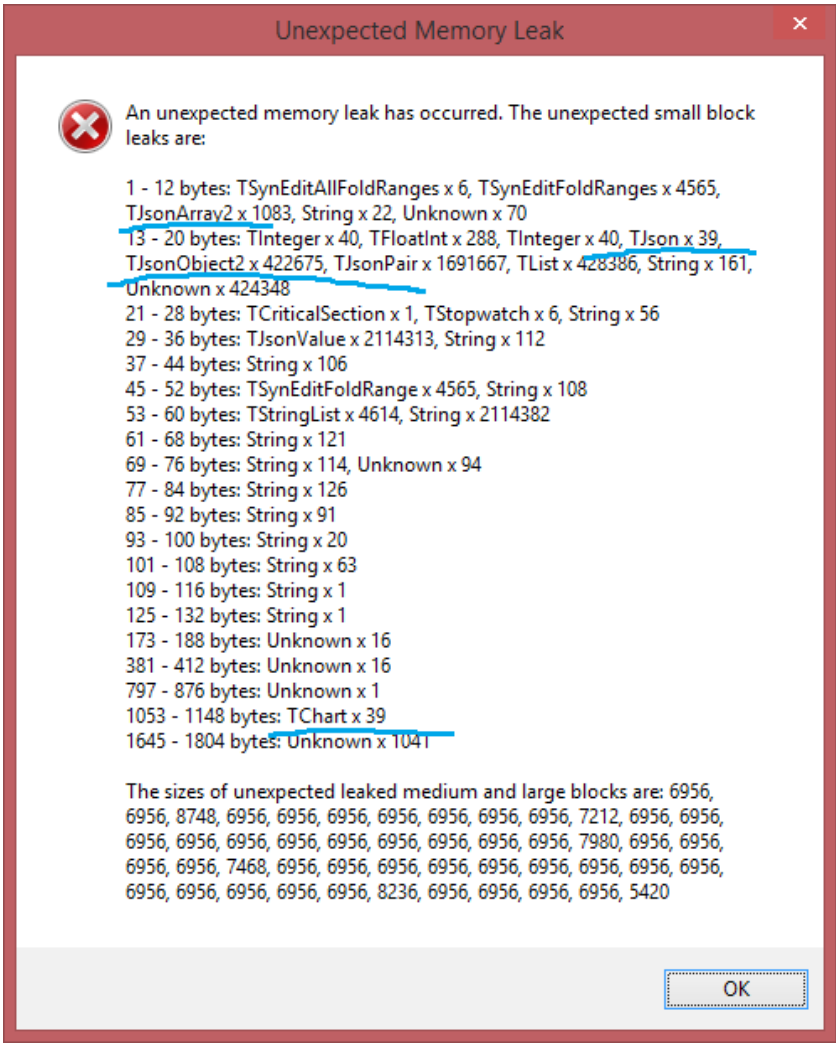
```
      iterate: rows
      distance
      duration
```

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 its important to know your memory allocation and free them as many as possible or keep the object lifetime short:



pic: m85_1026_dataleakmark.png

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...
```

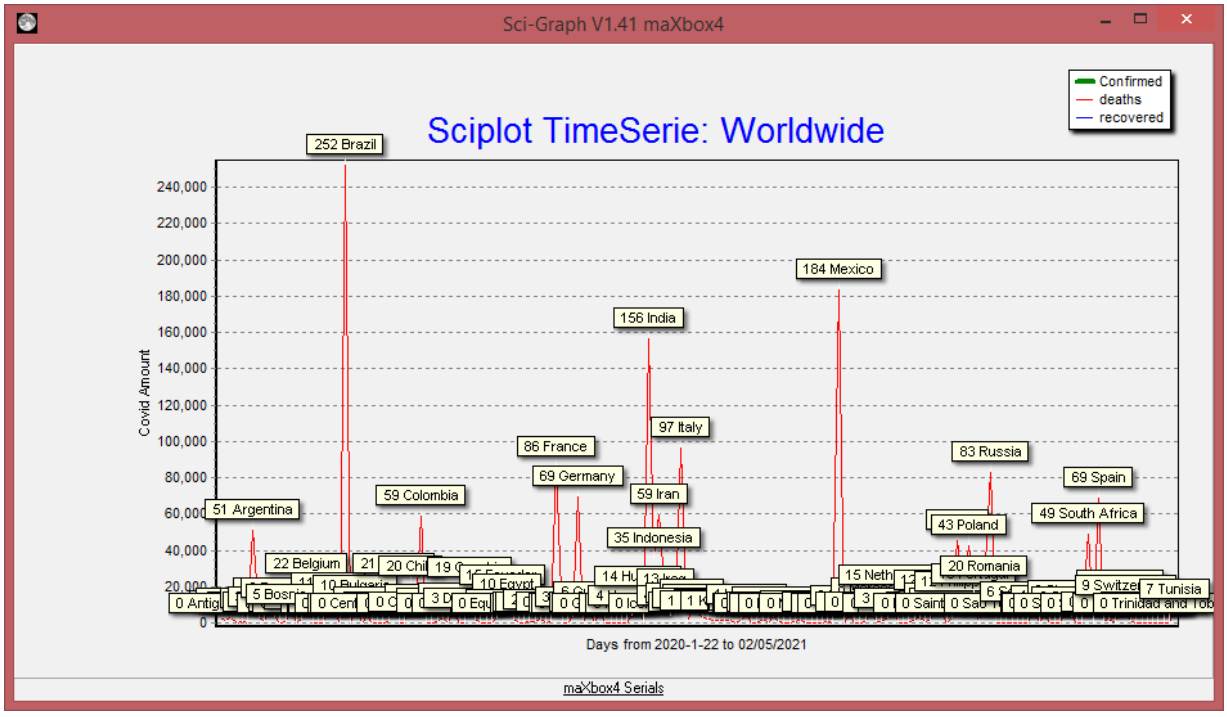
So the country is an object to get. Ok, it is a JsonObject dictionary with 192 countries. We check the keys of our dict with a nested loop of all confirmed cases:

```
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:

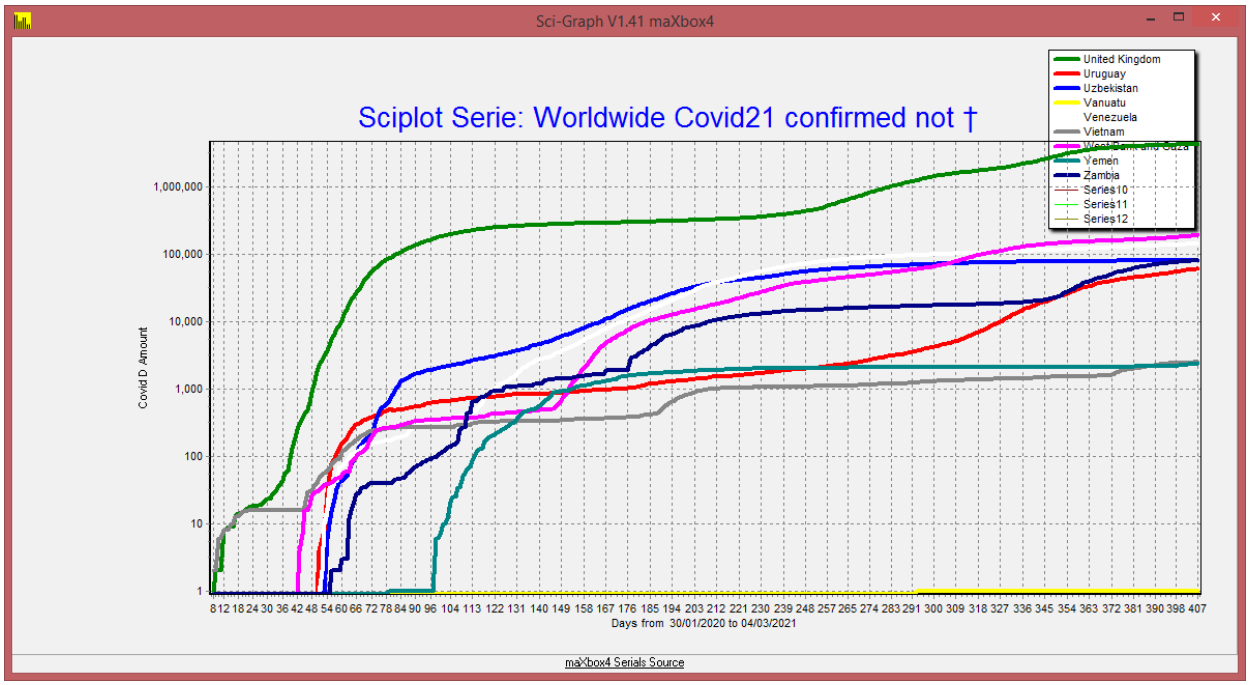
```
//*)  //accumulated
   Clabel:='Vietnam';
   Chart1.Title.Text.clear;
   Chart1.Title.Text.add('Sciplot TimeSerie for: '+Clabel);

     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;
```



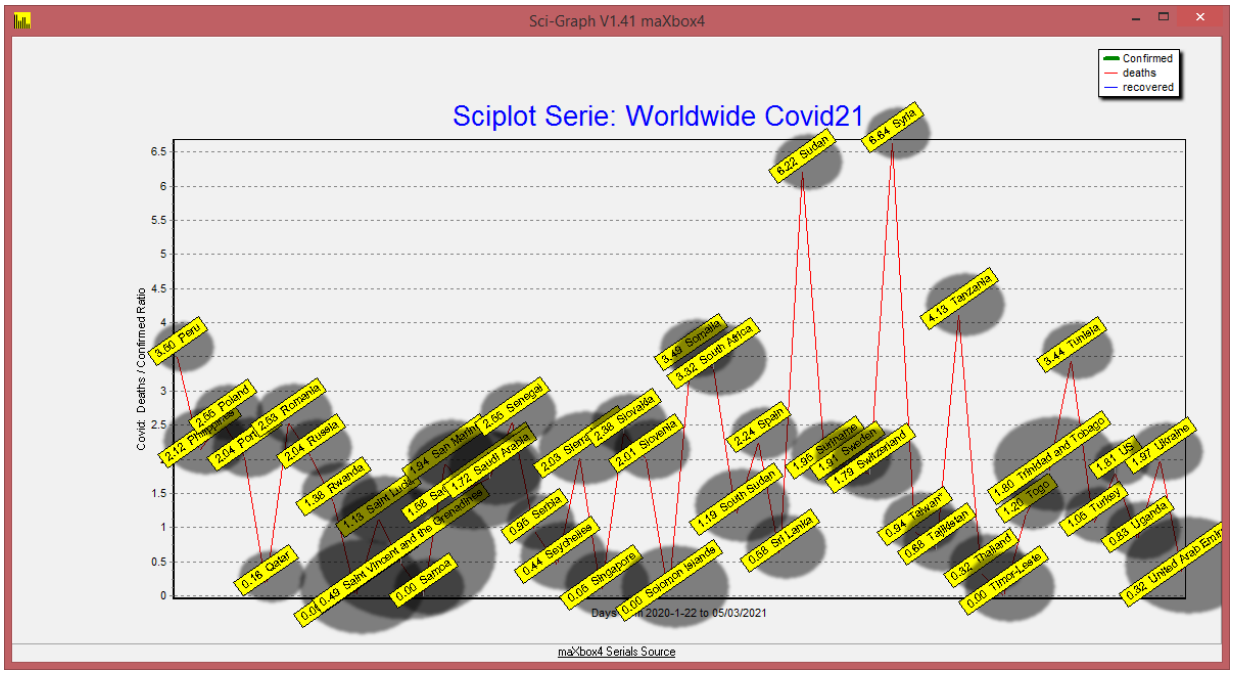
pic: m85_covid5_85.png

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.



Pic: m85_covid3.png

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.



Pic: m85_deathconfratio_json.png

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

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;
```

maXbox4exe Digitally signed by maXbox4exe
Date: 2021.05.17 13:44:01 +02'00'