White Paper

embarcadero®

Learn How to Use the New JSON Features in RAD Studio 10 "Seattle"

Paweł Głowacki

Embarcadero Technical Lead for Developer Tools

March 2016

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

EXECUTIVE SUMMARY

RAD Studio 10 "Seattle" introduces many new features for working with JSON documents. In this paper you will learn new ways of reading and generating JSON that require less memory and provide better performance.

JSON, or "Java Script Object Notation", is one of the most widely used light-weight data interchange formats, which is simple and human-readable.

Delphi and C++Builder, which are part of RAD Studio, have been providing support for JSON for many years. In the latest RAD Studio 10 "Seattle" release there are new ways of generating and parsing JSON based on streaming, rather than building JSON representation in memory.

In this whitepaper we will start from reviewing JSON basics and then we are going to move to discussing existing and new 10 "Seattle" support for working with JSON.

In the last part of this paper we are going to see how you can extend JSON support in RAD Studio 10 "Seattle" with custom "TJSONDocument" and "TJSONTreeView" components that we are going to use to build a simple JSON viewer utility application.

TABLE OF CONTENTS

INTRODUCTION

The new RAD Studio 10 "Seattle" brings a lot of new and improved features to both Delphi and C++Builder. There are big things like the brand new CLANG-based C++11 Win32 compiler or new support for MongoDB, but there are also many smaller new features and improvements.

If I would need to choose just one single new feature in RAD Studio 10 "Seattle" that would be the new support for JSON processing. JSON is currently the most important text format for data interchange. In the past we had XML, but it became a monster with tons of technologies around it like namespaces, schemas and more. XML parser is no longer a trivial piece of code. On the other hand one of the key design objectives of JSON (https://www.ietf.org/rfc/rfc4627.txt) - or more formally "Java Script Object Notation" - is to remain simple, but yet being able to express complex data structures.

Before RAD Studio 10, the JSON support was based on the "Document Object Model" where you could represent JSON with objects created in memory.

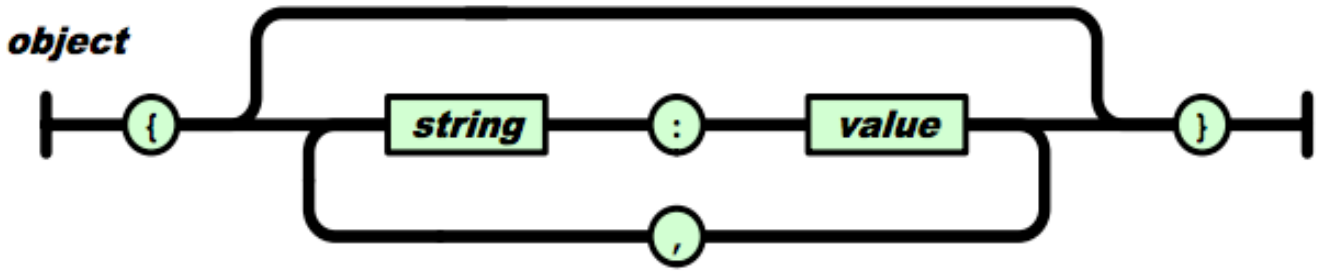
RAD Studio 10 "Seattle" introduces brand new JSON framework that is based not on "Document Object Model" but rather resembles XML SAX parsers where processing is based on streams and is much faster and consumes less memory.

WHAT IS JSON?

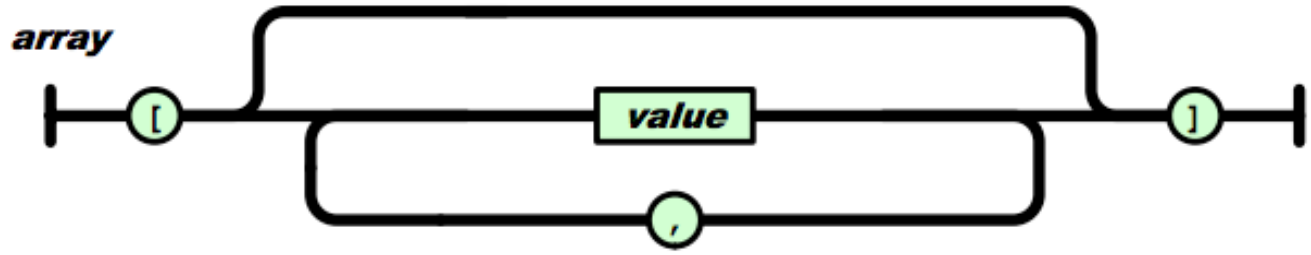
The http://json.org has the definition of the JSON syntax in the form of "railway" diagrams on just one short page.

JSON is built on just two structures: objects and arrays.

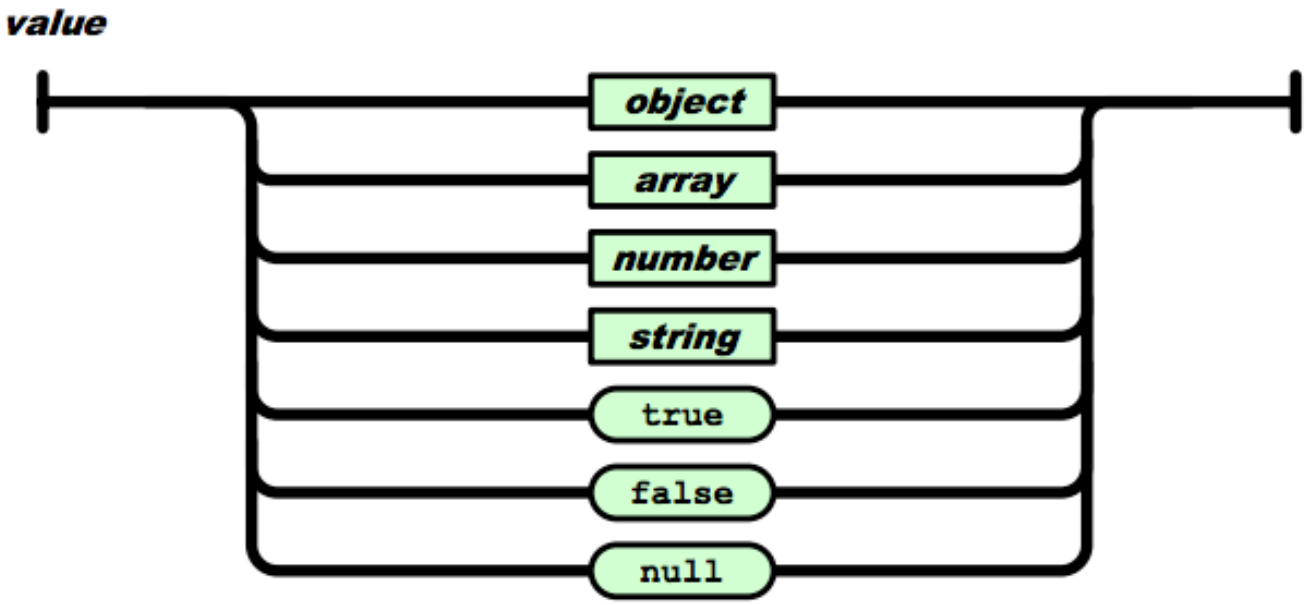
A JSON object is an unordered set of name-value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).



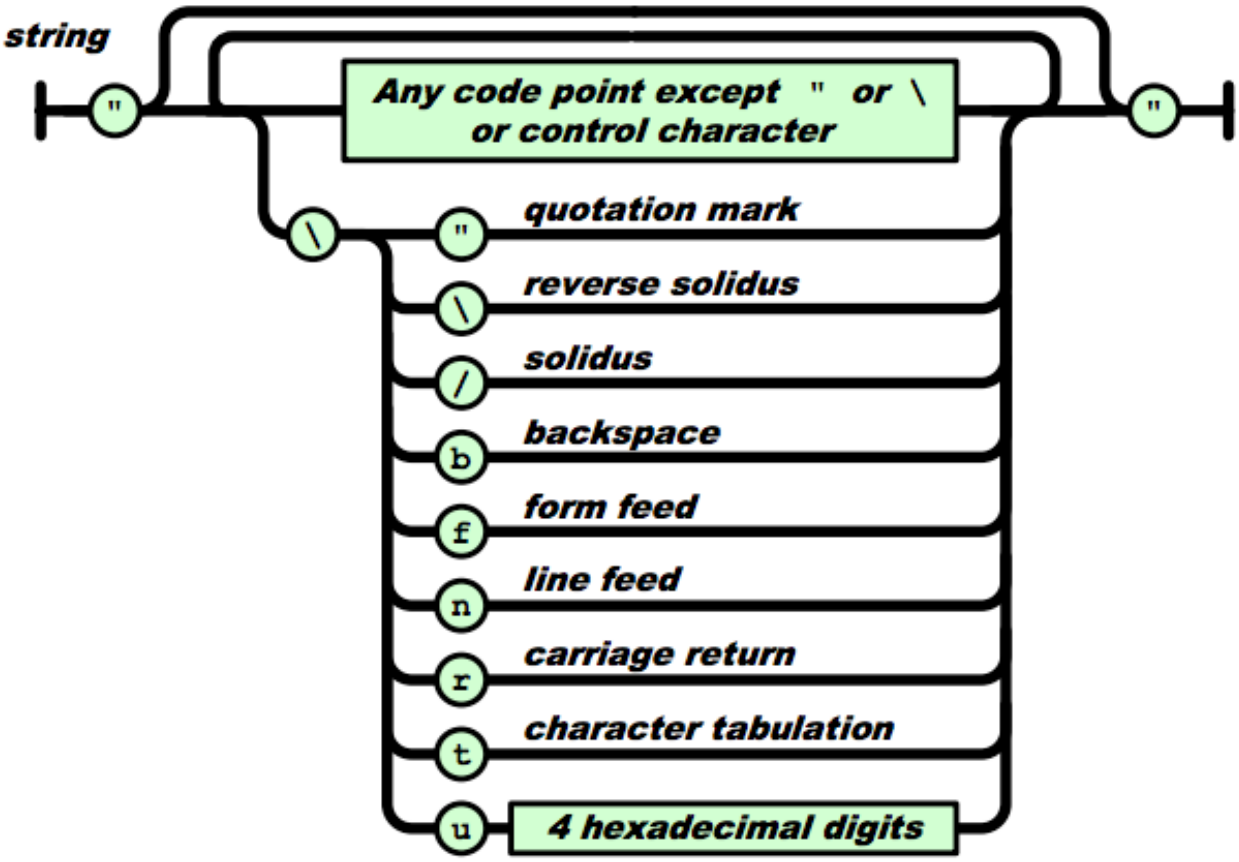
A JSON array is an ordered collection of values. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma).



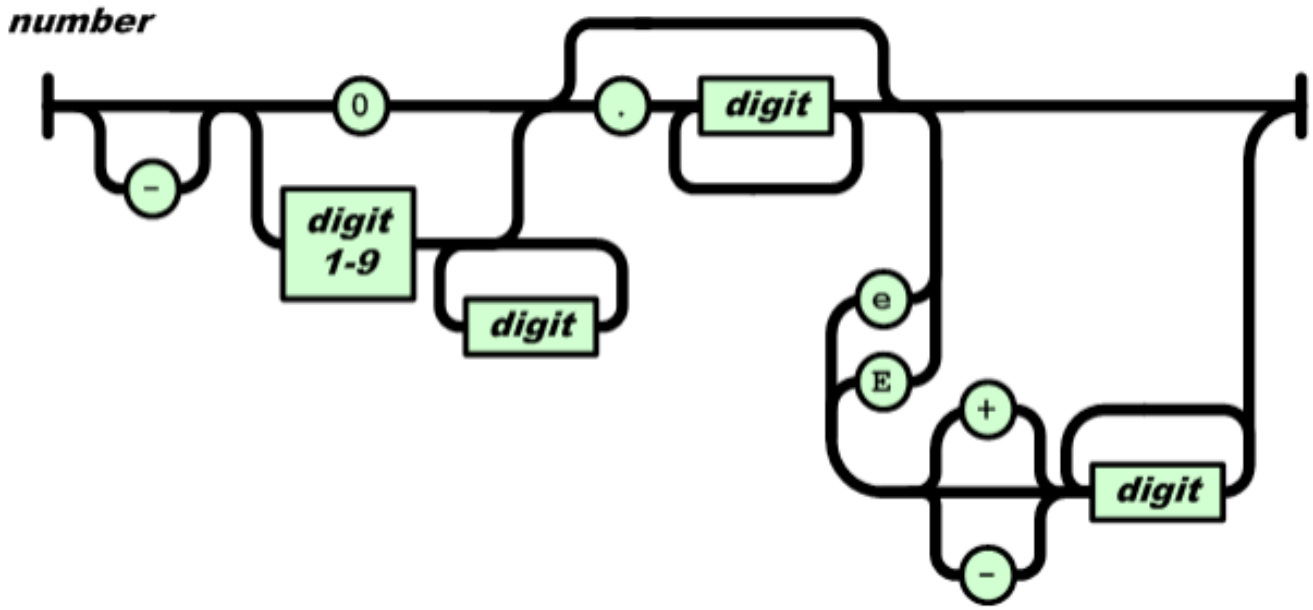
The key to JSON flexibility is a concept of a JSON "value". A value can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.



A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.



A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used.



Whitespace can be inserted between any pair of tokens. Except for a few encoding details that completely describes the JSON language.

This simplicity is key! JSON specification is very elegant and there are mappings from JSON to most of the existing programming languages, which makes it a great choice for a powerful data interchange format.

Let's have a look at how JSON concepts map into Delphi and C++Builder that are part of RAD Studio.

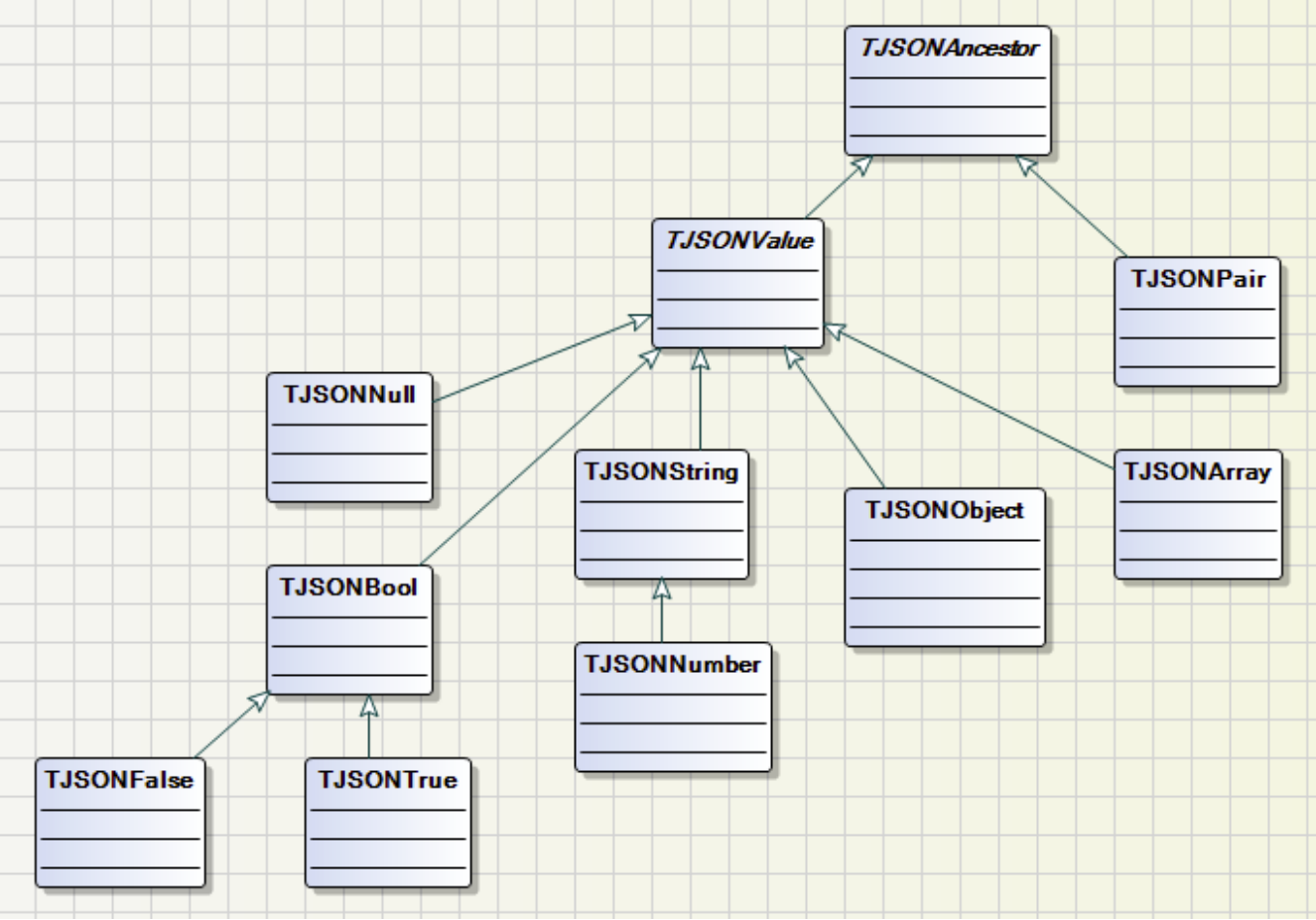
JSON SUPPORT IN RAD STUDIO

RAD Studio provides developers with technology to build desktop and mobile applications in either Object Pascal language used in Delphi or in standard C++ used in C++Builder.

JSON support is provided on a very low level of runtime library ("RTL"), so you can use JSON in all kinds of projects that RAD Studio supports, including:

- Delphi and C++Builder
- Windows VCL and Multi-Device FireMonkey projects

Core classes for working with JSON are defined in "System.JSON" unit (Delphi) or "System::JSON" namespace (C++). The hierarchy of these classes resembles JSON specification, where all simple and complex types inherit from a "value". In this way it is possible to use JSON to represent complex data structures.



RAD Studio 10 "Seattle" introduces new ways of working with JSON. It is still possible to work with JSON using its in-memory, graph representation using JSON objects from "System.JSON", but one can also use the new "streaming" model, that unlike "Document Object Model", does not require creating temporary objects in memory. The new "streaming" model provides abstract base classes for writing ("TJsonWriter") and reading ("TJsonReader") JSON. The actual JSON processing is done with specialized inherited classes like "TJsonTextWriter" and "TJsonTextReader".

There are also the new "fluent builder" classes. With "builders" you can write code that is very readable and resembles the structure of resulting JSON.

Another big new feature in RAD Studio 10 "Seattle" is the support for working with MongoDB NoSQL database which internal data format is JSON, or more accurately, "BSON" (binary JSON).

WRITING JSON

In 10 "Seattle" we have three different possible approaches to writing JSON. The old way ("DOM"), where we build JSON representation in memory using classes from "System.JSON" unit, or two new ways: with "TJsonTextWriter" or "TJsonObjectBuilder".

Let's consider a very simple piece of JSON. A hypothetical information about value of different stocks. To keep the sample simple we are going to have just two stock records, with just a symbol of a stock and its price.

```
{
  "Stocks": [
    {
      "symbol": "ACME",
      "price": 75.5
    },
    {
      "symbol": "COOL",
      "price": 21.7
    }
  ]
}
```

Let's build a simplistic multi-device application with just a memo for outputting JSON and three buttons on a toolbar for trying these three approaches of generating JSON.

Here is the snippet of code that demonstrates how to write JSON with DOM. Make sure to add "System.JSON" to the "uses" clause of your form.

```
uses
  System.JSON;

procedure TFormJsonWrite.ButtonDOMClick(Sender: TObject);
var
  objStocks, objS1, objS2: TJSONObject;
  arrStocks: TJSONArray;

begin
  objStocks := TJSONObject.Create;
  try
    arrStocks := TJSONArray.Create;

    objS1 := TJSONObject.Create;
    objS1.AddPair('symbol', TJSONString.Create('ACME'));
    objS1.AddPair('price', TJSONNumber.Create(75.5));

    arrStocks.Add(objS1);

    objS2 := TJSONObject.Create;
    objS2.AddPair('symbol', TJSONString.Create('COOL'));
    objS2.AddPair('price', TJSONNumber.Create(21.7));

    arrStocks.Add(objS2);

    objStocks.AddPair('Stocks', arrStocks);

    MemoLog.Lines.Clear;
    MemoLog.Lines.Add(objStocks.ToString);

  finally
    objStocks.Free;
  end;
end;
```

In this approach we need to define local variables for JSON objects and a JSON array to build a graph of objects in memory and then output it with "ToString" method of the root "objStocks" instance. Note that we only need to free the root object. All other objects are owned by the root object and if you try to free them in code, you would get an error.

The second approach involves using "TJsonTextWriter" class. In the "OnClick" event for the second button add the following code that generates the very same JSON as in the first example.

```
uses
  System.JSON,
  System.JSON.Types,
  System.JSON.Writers;

procedure TFormJsonWrite.ButtonWriterClick(Sender: TObject);
var
  StringWriter: TStringWriter;
  Writer: TJsonTextWriter;
begin
  StringWriter := TStringWriter.Create();
  Writer := TJsonTextWriter.Create(StringWriter);
  try
    Writer.Formatting := TJsonFormatting.Indented;

    Writer.WriteStartObject;
    Writer.WritePropertyName('Stocks');
    Writer.WriteStartArray;

    Writer.WriteStartObject;
    Writer.WritePropertyName('symbol');
    Writer.WriteValue('ACME');
    Writer.WritePropertyName('price');
    Writer.WriteValue(75.5);
    Writer.WriteEndObject;

    Writer.WriteStartObject;
    Writer.WritePropertyName('symbol');
    Writer.WriteValue('COOL');
    Writer.WritePropertyName('price');
    Writer.WriteValue(21.7);
    Writer.WriteEndObject;

    Writer.WriteEndArray;
    Writer.WriteEndObject;

    MemoLog.Lines.Clear;
    MemoLog.Lines.Add(StringWriter.ToString);

  finally
    Writer.Free;
    StringWriter.Free;
  end;

end;
```

The constructor of the "TJsonTextWriter" class expects as its argument an instance of "TTextWriter" that will be responsible for the actual process of outputting the resulting JSON text. Notice that the "TTextWriter" class is a class with virtual abstract methods that just define the interface to the text writing functionality so we need to use one of the text writer descendants like "TStringWriter". This is a very elegant approach and make it easy to use other types of writers without changing the logic for JSON writing.

Before calling different "write*" methods we set the "Formatting" property of the writer to "TJsonFormatting.Indented", so the resulting JSON will be nicely formatted. Note that in the DOM approach we just got JSON in one, not formatted string. The code to write JSON with text writer is much more verbose and does not allocate any temporary objects.

The third approach to write JSON involves using "fluent builders". The idea is to make the code even more verbose and more resembling the resulting JSON.

```
uses
  System.JSON,
  System.JSON.Types,
  System.JSON.Writers,
  System.JSON.Builders;

procedure TFormJsonWrite.ButtonBuilderClick(Sender: TObject);
var
  StringWriter: TStringWriter;
  Writer: TJsonTextWriter;
  Builder: TJSONObjectBuilder;

begin
  StringWriter := TStringWriter.Create();
  Writer := TJsonTextWriter.Create(StringWriter);
  Builder := TJSONObjectBuilder.Create(Writer);
  try
    Writer.Formatting := TJsonFormatting.Indented;

    Builder
    .BeginObject
      .BeginArray('Stocks')
        .BeginObject
          .Add('symbol', 'ACME')
          .Add('price', 75.5)
        .EndObject
        .BeginObject
          .Add('symbol', 'COOL')
          .Add('price', 21.7)
        .EndObject
      .EndArray
    .EndObject;
```

```
    MemoLog.Lines.Clear;
    MemoLog.Lines.Add(StringWriter.ToString);

  finally
    Builder.Free;
    Writer.Free;
    StringWriter.Free;
  end;
end;
```

As in the previous example we still need "TStringWriter" and "TJsonTextWriter" instances, but on top of them we need to instantiate "TJsonObjectBuilder" that accepts "TJsonTextWriter" as argument to its constructor.

The code to write JSON is one long chain of method calls on the builder class instance. Just from the structure of source code we can visualize what kind of JSON we are going to generate. That's the ultimate verbosity!

RAD Studio 10 "Seattle" comes with a very nice demo "JsonWorkbench" that is installed by default to "C:\Users\Public\Documents\Embarcadero\Studio\17.0\Samples\Object Pascal\RTL\Json" directory. With this demo you can interactively generate JSON writer Object Pascal code. Just paste a sample of JSON into the memo control at the left side of the screen. This demo also demonstrates converting JSON to BSON ("binary JSON") and back.

READING JSON

There is also the new way of reading JSON that is very similar to writer approach. You can still read JSON in the old way, by traversing the JSON object representation in memory, but it is faster and more efficient to use a reader to process the JSON stream directly.

For simplicity we are going to use the same JSON "Stocks" sample.

Again let's start with creating a new Delphi Multi-device project. Add a toolbar to the form. Add a memo, align it to the left and paste the sample JSON that we are going to read. Rename the memo to "MemoSrc".

First we are going to implement a convenient private method that will return a JSON string from the contents of the "MemoSrc".

```
function TFormJsonRead.GetJsonText: string;
var s: string;
begin
  for s in MemoSrc.Lines do
    Result := Result + s;
end;
```

Add a TListBox component to the form. Align it to the left and rename it to "ListBoxTokens". Drop a button on the toolbar and change its caption to "Read Tokens". Write the following code in the body of the "OnClick" event handler for the button. Also make sure to add "System.JSON.Readers" and "System.JSON.Types" to the "uses" clause in the "interface" section of the form.

```
procedure TFormJsonRead.ButtonReadTokensClick(Sender: TObject);
var
  jsontext, s: string; sr: TStringReader; jtr: TJsonTextReader;

begin
  ListBoxTokens.Items.Clear;
  jsontext := GetJsonText;

  sr := TStringReader.Create(jsontext);
  try
    jtr := TJsonTextReader.Create(sr);
    try
      while jtr.Read do
```

```
      begin
        s := JsonTokenToString(jtr.TokenType);
        ListBoxTokens.Items.Add(s);
      end;
    finally
      jtr.Free;
    end;
  finally
    sr.Free;
  end;

end;
```

The process of reading JSON is very similar to writing. There is "TJsonTextReader" class that expects as a parameter "TTextReader" class reference. The "TTextReader" is an abstract class, so that is why we need to start from building an instance of one of its descendants, which is in our case is the "TStringReader" that takes as argument "jsontext" string variable with JSON that we want to read.

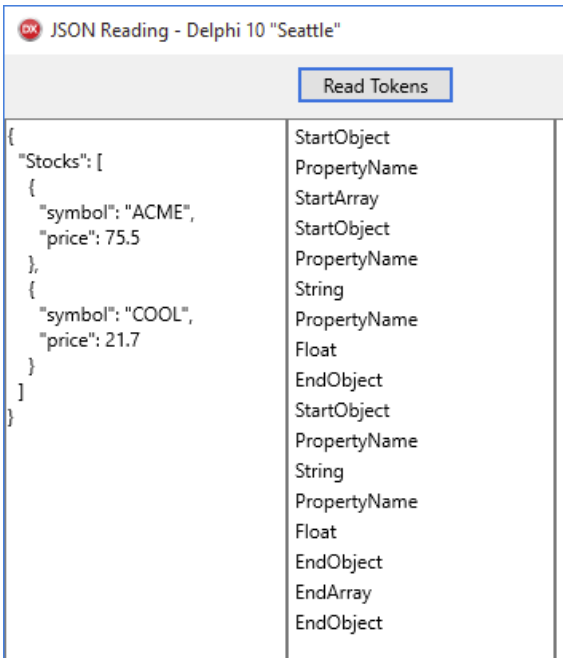
"TJsonTextReader" class is responsible for tokenizing JSON text. Calling the "Read" method advances the reader in the input stream of JSON tokens and returns "true" until there are more tokens to read or "false" when we are at the end of the stream. JSON reader class has the property "TokenType" that we can read to check what type of token we have just read. "TJsonToken" is an enumerated type defined in "System.JSON.Types" unit. In order to display token type in the listbox we need to implement "JsonTokenToString" routine.

Add to the project a new empty unit, save it as "JsonUtils" and enter there the following code:

```
unit JsonUtils;

interface

uses
  System.JSON.Types;

function JsonTokenToString(const t: TJsonToken): string;

implementation

function JsonTokenToString(const t: TJsonToken): string;
begin
  case t of
    TJsonToken.None: Result := 'None';
    TJsonToken.StartObject: Result := 'StartObject' ;
    TJsonToken.StartArray: Result := 'StartArray' ;
```

```
    TJsonToken.StartConstructor: Result := 'StartConstructor' ;
    TJsonToken.PropertyName: Result := 'PropertyName' ;
    TJsonToken.Comment: Result := 'Comment' ;
    TJsonToken.Raw: Result := 'Raw' ;
    TJsonToken.Integer: Result := 'Integer' ;
    TJsonToken.Float: Result := 'Float' ;
    TJsonToken.String: Result := 'String' ;
    TJsonToken.Boolean: Result := 'Boolean' ;
    TJsonToken.Null: Result := 'Null' ;
    TJsonToken.Undefined: Result := 'Undefined' ;
    TJsonToken.EndObject: Result := 'EndObject' ;
    TJsonToken.EndArray: Result := 'EndArray' ;
    TJsonToken.EndConstructor: Result := 'EndConstructor' ;
    TJsonToken.Date: Result := 'Date' ;
    TJsonToken.Bytes: Result := 'Bytes' ;
    TJsonToken.Oid: Result := 'Oid' ;
    TJsonToken.RegEx: Result := 'RegEx' ;
    TJsonToken.DBRef: Result := 'DBRef' ;
    TJsonToken.CodeWScope: Result := 'CodeWScope' ;
    TJsonToken.MinKey: Result := 'MinKey' ;
    TJsonToken.MaxKey: Result := 'MaxKey' ;
  end;
end;

end.
```

Run the application and click on the button. You should see the list of JSON tokens.



That's a good starting point to reading JSON, however in a typical scenario we would like to convert JSON text into something that we could manipulate in code. For example we would like to convert JSON to a list of custom objects that encapsulate individual stock symbols and their prices.

Let's define a very simple class for a stock record. Add to the project a new unit and name it "uCustomTypes". Enter the following code into this unit.

```
unit uCustomTypes;

interface

uses
  System.Generics.Collections;

type
  TStock = class
  private
    FSymbol: string;
    FPrice: double;
    procedure SetPrice(const Value: double);
    procedure SetSymbol(const Value: string);
  public
    property Symbol: string read FSymbol write SetSymbol;
    property Price: double read FPrice write SetPrice;
  end;

  TStocks = TObjectList<TStock>;

implementation

{ TStock }

procedure TStock.SetPrice(const Value: double);
begin
  FPrice := Value;
end;

procedure TStock.SetSymbol(const Value: string);
begin
  FSymbol := Value;
end;

end.
```

There is also a definition of "TStocks" class that is just a generic list of "TStock" objects.

The next objective is to convert the JSON text from the memo into an instance of "TStocks" class that can be used in code for arbitrary purposes.

Now we are going to implement two different ways of reading JSON and converting it into "TStocks" instance. In the new way, with a JSON reader, and in the old way using DOM and traversing the graph of JSON objects in memory.

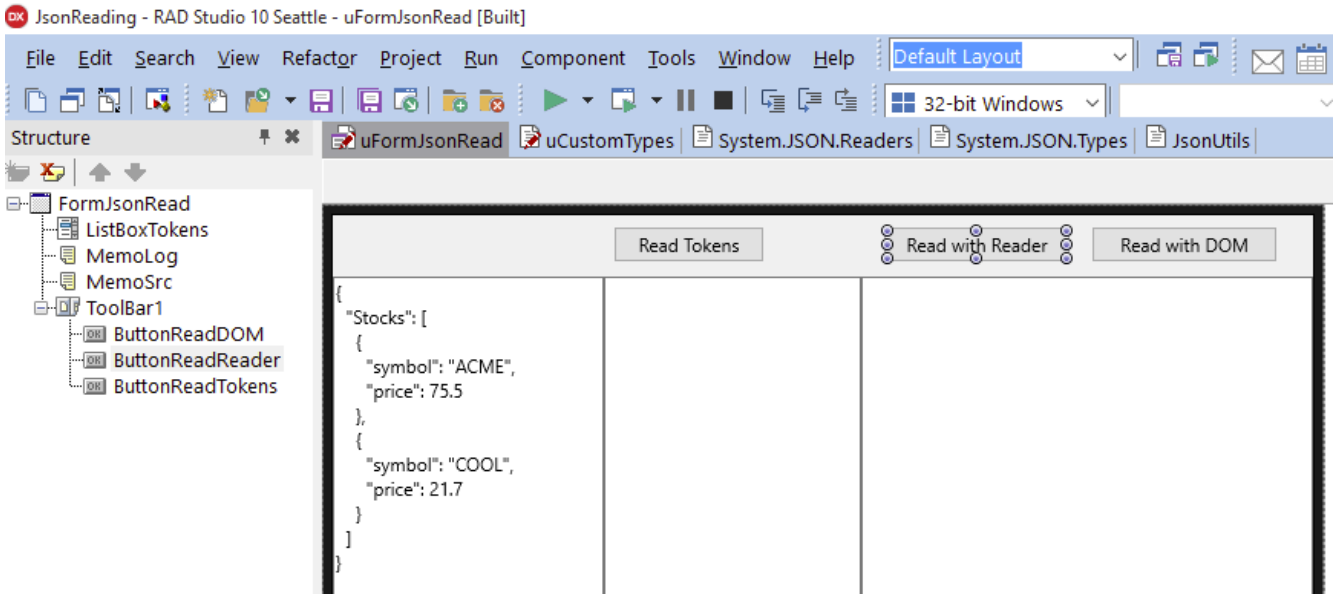
First we would like to have a way of visualizing the contents of "TStocks" instance to verify that the JSON was read correctly. For this we are going to add to the form another TMemo control. Rename it to "MemoLog". We are going to implement "DisplayStocks" method that takes as a parameter "TStocks" reference and outputs its contents to the form. We will also need a very simple "Log" method that takes a string and adds it to "MemoLog".

Declare "DisplayStocks(stocks: TStocks)" method in the private section of the form class declaration, invoke code completion with "Ctrl-Shift-C" key combination and enter the following code:

```
procedure TFormJsonRead.Log(s: string);
begin
  MemoLog.Lines.Add(s);
end;

procedure TFormJsonRead.DisplayStocks(stocks: TStocks);
var stock: TStock; i: integer;
begin
  MemoLog.Lines.Clear;

  Log(DateTimeToStr(Now));

  if stocks <> nil then
  begin
    Log('Stocks count = ' + stocks.Count.ToString);
    Log('==================');

    for i := 0 to stocks.Count-1 do
    begin
      stock := stocks[i];
      Log('Stock Nr: ' + i.ToString);
      Log('Symbol: ' + stock.Symbol);
      Log('Price: ' + stock.price.ToString);
      Log('==================');
    end;
  end
  else
    Log('No stocks found.');
end;
```

We are logging the current time as the first thing. This is because we are expecting exactly the same output from different methods of reading JSON, so we want to see when this output was generated.

Add two buttons to the toolbar. One for reading JSON with DOM and one for using the new JSON reader.



First let's implement reading JSON with DOM. In the "OnClick" event of the button add the following code:

```
procedure TFormJsonRead.ButtonReadDOMClick(Sender: TObject);
var stocks: TStocks; valRoot: TJSONValue; objRoot: TJSONObject;
  valStocks: TJSONValue; arrStocks: TJSONArray; i: integer;
begin
  stocks := TStocks.Create;
  try
    valRoot := TJSONObject.ParseJSONValue(GetJsonText);
    if valRoot <> nil then
    begin
      if valRoot is TJSONObject then
      begin
        objRoot := TJSONObject(valRoot);
        if objRoot.Count > 0 then
        begin
          valStocks := objRoot.Values['Stocks'];
          if valStocks <> nil then
          begin
            if valStocks is TJSONArray then
            begin
```

```
              arrStocks := TJSONArray(valStocks);
              for i := 0 to arrStocks.Count-1 do
              begin
                if arrStocks.Items[i] is TJSONObject then
                  ProcessStockObj(stocks, TJSONObject(arrStocks.Items[i]));
              end;
            end;
          end;
        end;
      end;
      valRoot.Free;
    end;

    DisplayStocks(stocks);
  finally
    stocks.Free;
  end;
end;

procedure TFormJsonRead.ProcessStockObj(stocks: TStocks; stockObj:
TJSONObject);
var stock: TStock; val: TJSONValue;
begin
  stock := TStock.Create;

  val := stockObj.Values['symbol'];
  if val <> nil then
    if val is TJSONString then
      stock.Symbol := TJSONString(val).Value;

  val := stockObj.Values['price'];
  if val <> nil then
    if val is TJSONNumber then
      stock.price := TJSONNumber(val).AsDouble;

  stocks.Add(stock);
end;
```

Note that there is a separate "ProcessStockObj" method that is called in the loop and encapsulates processing of each stock record.

The key to parsing JSON using DOM is the class method "TJSONObject.ParseJSONValue". This an overloaded method defined in "System.JSON" and it takes JSON text in form of a string, stream or array of bytes, parses it and returns "TJSONValue" reference which points to the root of a graph of JSON objects in memory. If JSON passed to this method is malformed then this method does not raise an exception, but just returns "nil".

Note that even for a simple JSON, like in our demo sample, the resulting code is quite complex and involves a lot of checking for "nil" and typecasting to expected types of JSON values. There is also an overhead of having to create a lot of in-memory objects that represent JSON.

The new way of reading JSON is based on "TJsonTextReader". We are going to implement the "OnClick" event to process individual stock records.

```
procedure TFormJsonRead.ButtonReadReaderClick(Sender: TObject);
var jtr: TJsonTextReader; sr: TStringReader; stocks: TStocks;
begin
  stocks := TStocks.Create;
  try
    sr := TStringReader.Create(GetJsonText);
    try
      jtr := TJsonTextReader.Create(sr);
      try
        while jtr.Read do
        begin
          if jtr.TokenType = TJsonToken.StartObject then
            ProcessStockRead(stocks, jtr);
        end;
      finally
        jtr.Free;
      end;
    finally
      sr.Free;
    end;

    DisplayStocks(stocks);
  finally
    stocks.Free;
  end;
end;

procedure TFormJsonRead.ProcessStockRead(stocks: TStocks; jtr:
TJsonTextReader);
var stock: TStock;
begin
  stock := TStock.Create;

  while jtr.Read do
  begin
    if jtr.TokenType = TJsonToken.PropertyName then
    begin
      if jtr.Value.ToString = 'symbol' then
      begin
        jtr.Read;
        stock.Symbol := jtr.Value.AsString;
      end
```

```
      else if jtr.Value.ToString = 'price' then
      begin
        jtr.Read;
        stock.price := jtr.Value.AsExtended;
      end
    end

    else if jtr.TokenType = TJsonToken.EndObject then
    begin
      stocks.add(stock);
      exit;
    end;
  end;
end;
```

We no longer need to create temporary objects. We just move through the stream of JSON tokens. Notice that we do not need to encapsulate the structure of the whole JSON in this code. It could be part of a bigger structure. We just iterate through tokens and if we encounter the "BeginObject" token we can start separate processing in the specialized procedure that only cares about objects that represent a stock record and have properties with particular names.

Using JSON readers instead of DOM is faster and requires less memory and processing power.

CUSTOM JSON COMPONENTS

We have learned so far how to use the new JSON features in RAD Studio 10 "Seattle" for efficient writing and reading JSON documents. Let's have a look how we can extend the JSON support in RAD Studio. We are going to implement two custom JSON components:

- **TJSONDocument** – which encapsulates the functionality of JSON parsing and provides access to JSON at design-time. This component is non-visual, so it can be used in either VCL or FireMonkey projects.

- **TJSONTreeView** – is a visual component for visualizing JSON in the form of a tree view. This component works with "TJSONDocument" and is a descendant of the VCL "TTreeView" component, so it can only be used in VCL projects.

The "TJSONDocument" component is a direct descendant of "TComponent" class. It has "JsonText" published string property. As a side effect of assigning to "JsonText" property, the JSON string is parsed and its DOM representation is available through the "RootValue: TJSONValue" public read-only property. "EstimatedByteSize" property can be used to get the size of parsed JSON data in bytes. Below is the complete source code of the "TJSONDocument".

```
unit JSONDoc;

interface

uses
  System.Classes, System.JSON;

type
  TJSONDocument = class(TComponent)
  private
    FRootValue: TJSONValue;
    FJsonText: string;
    FOnChange: TNotifyEvent;
    procedure SetJsonText(const Value: string);
  protected
    procedure FreeRootValue;
    procedure DoOnChange; virtual;
    procedure ProcessJsonText;
  public
    class function StripNonJson(s: string): string; inline;
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function IsActive: boolean;
    function EstimatedByteSize: integer;
```

```
    property RootValue: TJSONValue read FRootValue;
  published
    property JsonText: string read FJsonText write SetJsonText;
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  end;

implementation

uses
  System.SysUtils, System.Character;

{ TJSONDocument }

constructor TJSONDocument.Create(AOwner: TComponent);
begin
  inherited;
  FRootValue := nil;
  FJsonText := '';
end;

destructor TJSONDocument.Destroy;
begin
  FreeRootValue;
  inherited;
end;

procedure TJSONDocument.FreeRootValue;
begin
  if Assigned(FRootValue) then
    FreeAndNil(FRootValue);
end;

procedure TJSONDocument.DoOnChange;
begin
  if Assigned(FOnChange) then
    FOnChange(self);
end;

function TJSONDocument.EstimatedByteSize: integer;
begin
  if IsActive then
    Result := FRootValue.EstimatedByteSize
  else
    Result := 0;
end;

function TJSONDocument.IsActive: boolean;
begin
  Result := RootValue <> nil;
end;

procedure TJSONDocument.SetJsonText(const Value: string);
```

```
begin
  if FJsonText <> Value then
  begin
    FreeRootValue;
    FJsonText := Value;
    if FJsonText <> '' then
      ProcessJsonText;
    if not IsActive then
      FJsonText := '';
  end;
end;

procedure TJSONDocument.ProcessJsonText;
var s: string;
begin
  FreeRootValue;
  s := StripNonJson(JsonText);
  FRootValue := TJSONObject.ParseJSONValue(BytesOf(s),0);
  DoOnChange;
end;

class function TJSONDocument.StripNonJson(s: string): string;
var ch: char; inString: boolean;
begin
  Result := '';
  inString := false;
  for ch in s do
  begin
    if ch = '"' then
      inString := not inString;

    if ch.IsWhiteSpace and not inString then
      continue;

    Result := Result + ch;
  end;
end;

end.
```

The "TJSONTreeView" component is a descendant of the VCL "TTreeView" Windows control, so it can be only used in VCL applications. It has "JSONDocument" property hooking the two components together at design-time. If "RootValue" property of "JSONDocument" is not "nil", then the component builds its tree view representation.

There are also "VisibleChildrenCounts" and "VisibleByteSizes" boolean properties that control which additional information will be displayed in the tree view.

The setter for the "JSONDocument" property checks if the "JSONDocument" and "RootValue" references are not "nil". If this is the case it calls the internal "LoadJson" method that is responsible for building the tree view. This method traverses the graph of JSON objects and recursively call "ProcessPair" or "ProcessElement" methods.

Here is the full source code of the "TJSONTreeView" component.

```
unit JSONTreeView;

interface

uses
  System.Classes, System.SysUtils, System.JSON, jsondoc, Vcl.ComCtrls;

type
  EUnknownJsonValueDescendant = class(Exception)
    constructor Create;
  end;

  TJSONTreeView = class(TTreeView)
  private
    FJSONDocument: TJSONDocument;
    FVisibleChildrenCounts: boolean;
    FVisibleByteSizes: boolean;
    procedure SetJSONDocument(const Value: TJSONDocument);
    procedure SetVisibleChildrenCounts(const Value: boolean);
    procedure SetVisibleByteSizes(const Value: boolean);
    procedure ProcessElement(currNode: TTreeNode; arr: TJSONArray;
      aIndex: integer);
    procedure ProcessPair(currNode: TTreeNode; obj: TJSONObject;
      aIndex: integer);
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    class function IsSimpleJsonValue(v: TJSONValue): boolean; inline;
    class function UnQuote(s: string): string; inline;
    constructor Create(AOwner: TComponent); override;
    procedure ClearAll;
    procedure LoadJson;
  published
    property JSONDocument: TJSONDocument
      read FJSONDocument write SetJSONDocument;
    property VisibleChildrenCounts: boolean
      read FVisibleChildrenCounts write SetVisibleChildrenCounts;
    property VisibleByteSizes: boolean
      read FVisibleByteSizes write SetVisibleByteSizes;
  end;

implementation
```

```
{ TJSONTreeView }

procedure TJSONTreeView.ClearAll;
begin
  Items.Clear;
end;

constructor TJSONTreeView.Create(AOwner: TComponent);
begin
  inherited;
  FVisibleChildrenCounts := true;
  FVisibleByteSizes := false;
end;

class function TJSONTreeView.IsSimpleJsonValue(v: TJSONValue): boolean;
begin
  Result := (v is TJSONNumber)
    or (v is TJSONString)
    or (v is TJSONTrue)
    or (v is TJSONFalse)
    or (v is TJSONNull);
end;

procedure TJSONTreeView.LoadJson;
var v: TJSONValue; currNode: TTreeNode; i, aCount: integer; s: string;
begin
  ClearAll;

  if (JSONDocument <> nil) and JSONDocument.IsActive then
  begin
    v := JSONDocument.RootValue;

    Items.BeginUpdate;
    try
      Items.Clear;

      if IsSimpleJsonValue(v) then
        Items.AddChild(nil, UnQuote(v.Value))

      else
      if v is TJSONObject then
      begin
        aCount := TJSONObject(v).Count;
        s := '{}';
        if VisibleChildrenCounts then
          s := s + ' (' + IntToStr(aCount) + ')';
        if VisibleByteSizes then
          s := s + ' (' + IntToStr(v.EstimatedByteSize) + ' bytes)';
        currNode := Items.AddChild(nil, s);
        for i := 0 to aCount - 1 do
          ProcessPair(currNode, TJSONObject(v), i)
```

```
      end

      else
      if v is TJSONArray then
      begin
        aCount := TJSONArray(v).Count;
        s := '[]';
        if VisibleChildrenCounts then
          s := s + ' (' + IntToStr(aCount) + ')';
        if VisibleByteSizes then
          s := s + ' (' + IntToStr(v.EstimatedByteSize) + ' bytes)';
        currNode := Items.AddChild(nil, s);
        for i := 0 to aCount - 1 do
          ProcessElement(currNode, TJSONArray(v), i)
      end

      else
        raise EUnknownJsonValueDescendant.Create;

    finally
      Items.EndUpdate;
    end;

    FullExpand;
  end;
end;

procedure TJSONTreeView.ProcessPair(currNode: TTreeNode; obj: TJSONObject;
aIndex: integer);
var p: TJSONPair; s: string; n: TTreeNode; i, aCount: integer;
begin
  p := obj.Pairs[aIndex];

  s := UnQuote(p.JsonString.ToString) + ' : ';

  if IsSimpleJsonValue(p.JsonValue) then
  begin
    Items.AddChild(currNode, s + p.JsonValue.ToString);
    exit;
  end;

  if p.JsonValue is TJSONObject then
  begin
    aCount := TJSONObject(p.JsonValue).Count;
    s := s + ' {}';
    if VisibleChildrenCounts then
      s := s + ' (' + IntToStr(aCount) + ')';
    if VisibleByteSizes then
        s := s + ' (' + IntToStr(p.EstimatedByteSize) + ' bytes)';
    n := Items.AddChild(currNode, s);
    for i := 0 to aCount - 1 do
      ProcessPair(n, TJSONObject(p.JsonValue), i);
```

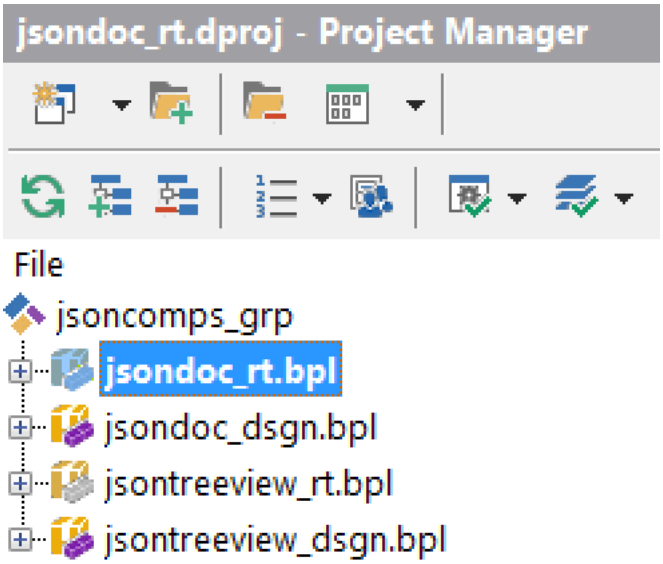
```
  end

  else if p.JsonValue is TJSONArray then
  begin
    aCount := TJSONArray(p.JsonValue).Count;
    s := s + ' []';
    if VisibleChildrenCounts then
      s := s + ' (' + IntToStr(aCount) + ')';
    if VisibleByteSizes then
        s := s + ' (' + IntToStr(p.EstimatedByteSize) + ' bytes)';
    n := Items.AddChild(currNode, s);
    for i := 0 to aCount - 1 do
      ProcessElement(n, TJSONArray(p.JsonValue), i);
  end
  else
    raise EUnknownJsonValueDescendant.Create;
end;

procedure TJSONTreeView.ProcessElement(currNode: TTreeNode; arr:
TJSONArray; aIndex: integer);
var v: TJSONValue; s: string; n: TTreeNode; i, aCount: integer;
begin
  v := arr.Items[aIndex];
  s := '[' + IntToStr(aIndex) + '] ';

  if IsSimpleJsonValue(v) then
  begin
    Items.AddChild(currNode, s + v.ToString);
    exit;
  end;

  if v is TJSONObject then
  begin
    aCount := TJSONObject(v).Count;
    s := s + ' {}';
    if VisibleChildrenCounts then
      s := s + ' (' + IntToStr(aCount) + ')';
    if VisibleByteSizes then
        s := s + ' (' + IntToStr(v.EstimatedByteSize) + ' bytes)';
    n := Items.AddChild(currNode, s);
    for i := 0 to aCount - 1 do
      ProcessPair(n, TJSONObject(v), i);
  end

  else if v is TJSONArray then
  begin
    aCount := TJSONArray(v).Count;
    s := s + ' []';
    n := Items.AddChild(currNode, s);
    if VisibleChildrenCounts then
      s := s + ' (' + IntToStr(aCount) + ')';
    if VisibleByteSizes then
```

```
        s := s + ' (' + IntToStr(v.EstimatedByteSize) + ' bytes)';
    for i := 0 to aCount - 1 do
      ProcessElement(n, TJSONArray(v), i);
  end
  else
    raise EUnknownJsonValueDescendant.Create;

end;

procedure TJSONTreeView.SetJSONDocument(const Value: TJSONDocument);
begin
  if FJSONDocument <> Value then
  begin
    FJSONDocument := Value;
    ClearAll;
    if FJSONDocument <> nil then
    begin
      if FJSONDocument.IsActive then
        LoadJson;
    end;
  end;
end;

procedure TJSONTreeView.SetVisibleByteSizes(const Value: boolean);
begin
  if FVisibleByteSizes <> Value then
  begin
    FVisibleByteSizes := Value;
    LoadJson;
  end;
end;

procedure TJSONTreeView.SetVisibleChildrenCounts(const Value: boolean);
begin
  if FVisibleChildrenCounts <> Value then
  begin
    FVisibleChildrenCounts := Value;
    LoadJson;
  end;
end;

class function TJSONTreeView.UnQuote(s: string): string;
begin
  Result := Copy(s,2,Length(s)-2);
end;

procedure TJSONTreeView.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited;

  if Operation = opRemove then
```

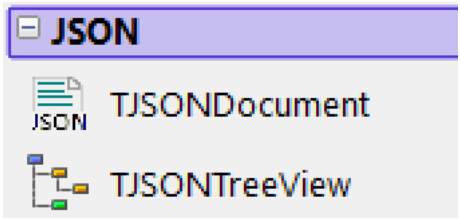
```
    if FJSONDocument <> nil then
      if AComponent = FJSONDocument then
      begin
        FJSONDocument := nil;
        ClearAll;
      end;
end;

{ EUnknownJsonValueDescendant }

resourcestring
  StrUnknownTJSONValueDescendant = 'Unknown TJSONValue descendant';

constructor EUnknownJsonValueDescendant.Create;
begin
  inherited Create(StrUnknownTJSONValueDescendant);
end;

end.
```

If you download the source code for these components, you will find the project group called "jsoncomps_grp" that contains four package projects. Each component has its own runtime and design-time package. The "TJSONDocument" component can be used in FireMonkey and VCL applications and "TJSONTreeView" only in VCL. That is the reason that they live in separate packages.

It's a good idea to put these components in a folder that is easy to find and not too deep in the file system. For example you can create a folder "C:\src" and extract there the contents of "JsonComponents" folder from the download. Open "jsoncomps_grp" project group in RAD Studio 10 "Seattle". You should see four package projects.

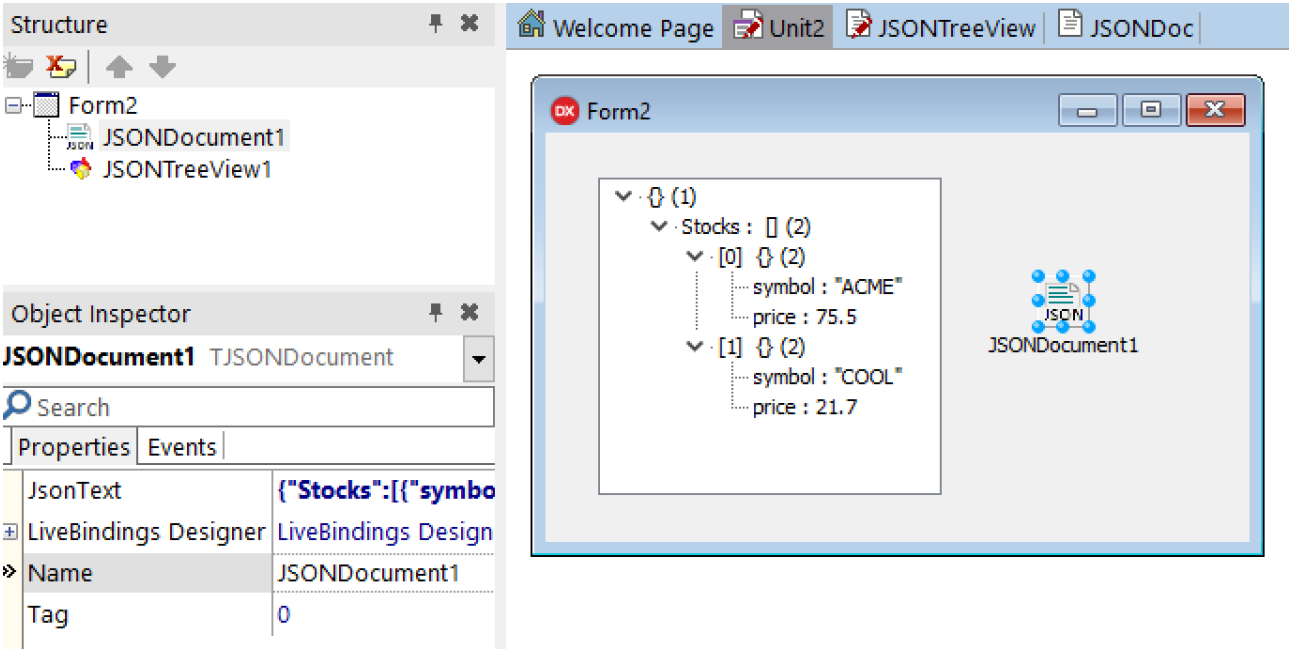


Right-click on the "jsoncomps_grp" node and select "Build All" from the context menu to compile all packages. Right-click on the "jsondoc_dsgn" node and select "Install". Do the same thing for the "jsontreeview_dsgn". You should get information messages that "TJSONDocument" and "TJSONTreeView" components have been successfully installed. You should be able to see them in the new "JSON" category in the RAD Studio "Tool Palette".



It is a good idea to add the folder where your components were extracted to to the "Browsing Path" of the RAD Studio, so you could jump to the source code of JSON components directly from within the editor. Select "Tools -> Options" from the main menu. Select "Delphi Options -> Library" in the dialog and add the path to components folder to the "Browsing Path" option.

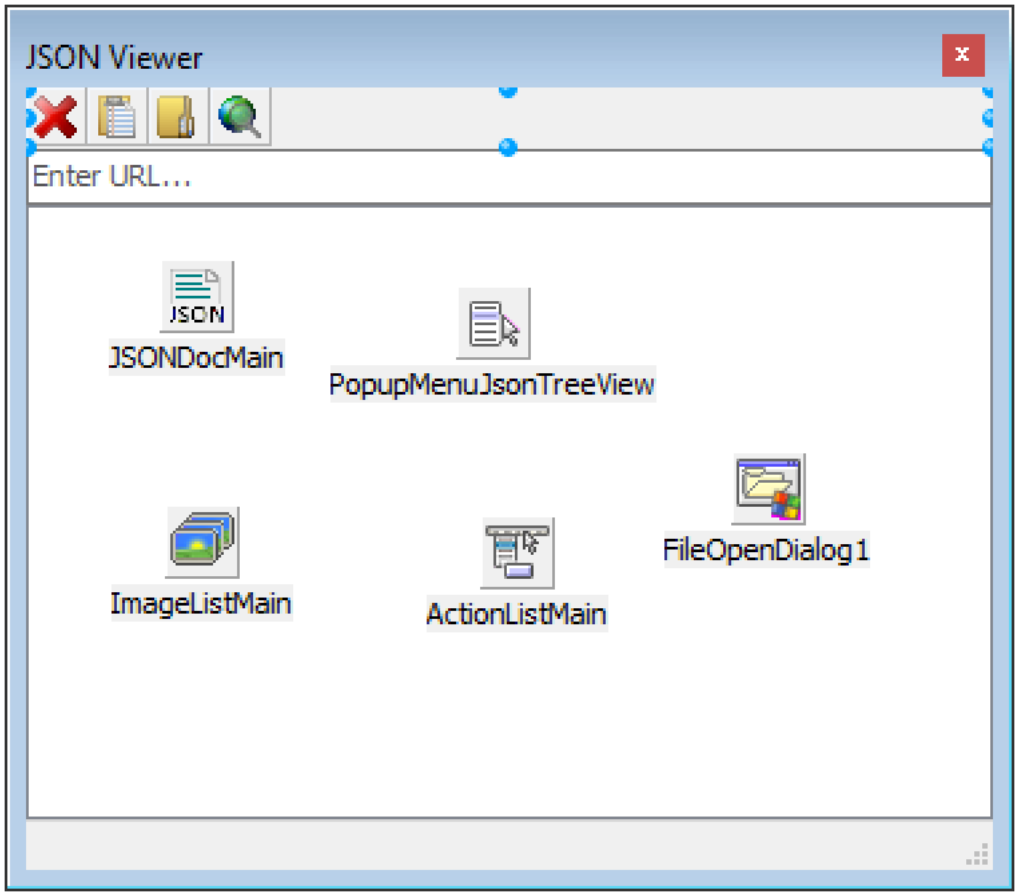
Right now we are ready to use the new JSON custom components. Create a new VCL Forms Application. Drop "TJSONDocument" on the form. Copy some JSON text to the clipboard, for example generated with our "JsonWriters" demo DOM option, and paste it into the "JsonText" property of the component. Now drop "TJSONTreeView" component onto the form and set its "JSONDocument" property to "JSONDocument1". At this moment you should see the JSON represented in the form of a tree view.



Feel free to use these components in your projects with no restrictions.

JSON VIEWER APP

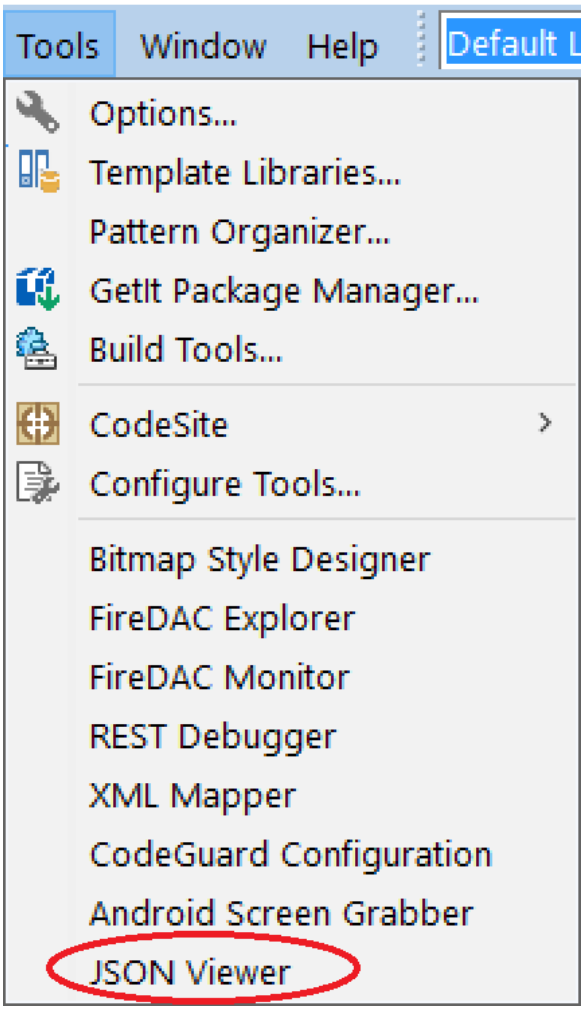
The last step is to use custom JSON components to build a simple utility application for displaying JSON coming from an URL, from a file or from a clipboard. The source code of this application is available for download and it is also available in a compiled form, so anybody can use it right away! Download locations are listed in "References" section of this document.



"JSONViewer" is a simple Delphi 10 "Seattle" VCL Forms project. It is using native HTTP client components ("TNetHTTPRequest" and "TNetHTTPClient") for downloading JSON from an arbitrary URL. There is also a context menu to choose JSON viewing options. It is possible to toggle on and off displaying JSON elements children counts and byte sizes.

The source code is very simple and self-explanatory, so I'm not going to list it here.

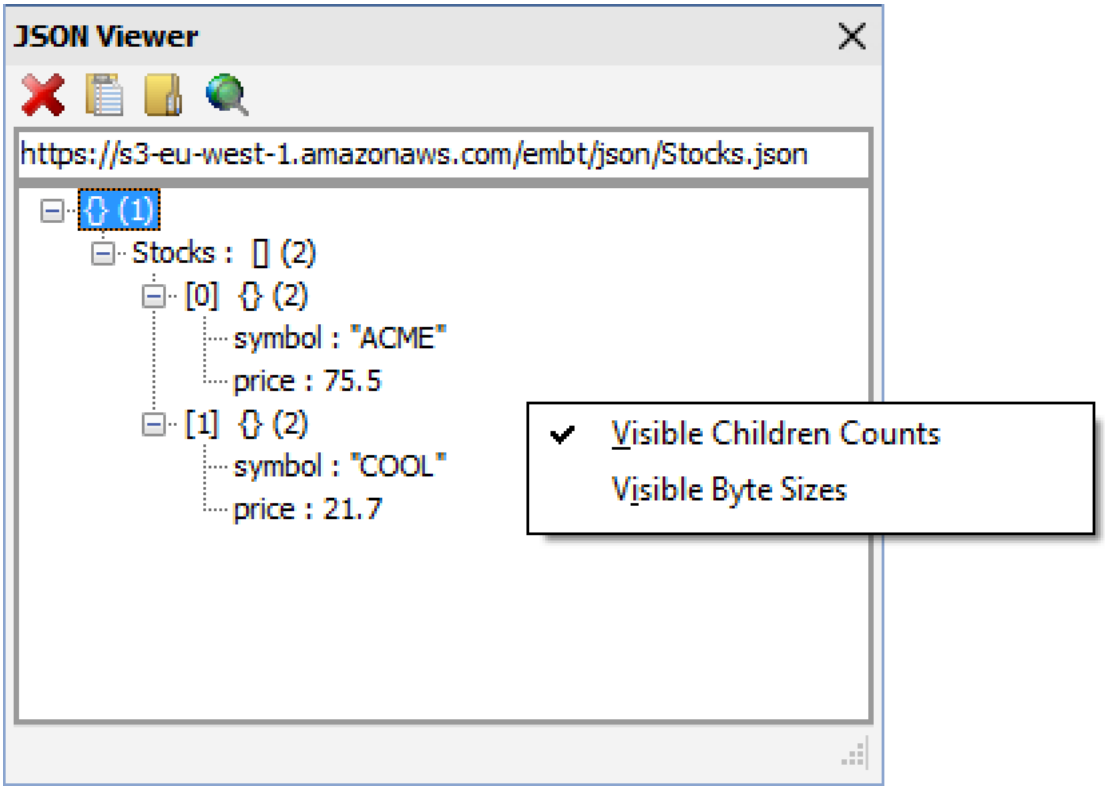
You can optionally add "JSON Viewer" to your "Tools" menu in the "IDE".



For testing purposes the sample "Stocks.json" file has been uploaded to AWS S3 bucket and you can read it from the following URL:

https://s3-eu-west-1.amazonaws.com/embt/json/Stocks.json

Here is the screenshot of Delphi 10 "JSONViewer" utility app in action:



I'm using this tool a lot. If you want to build an app that reads JSON from the Internet, you need to know its structure in advance to be able to write the code that will properly ready it!

SUMMARY

RAD Studio 10 "Seattle" is a great release with tons of new and improved features that Delphi and C++ developers enjoy in their day to day work. One of the most useful new RTL features are the new ways of writing and reading JSON.

JSON is one of the most important modern formats for data interchange between different systems and programming languages. In this paper we have reviewed basics of JSON specification and support for working with JSON in RAD Studio. We have compared existing in-memory support with the new ways of writing and reading JSON that provide better performance.

There are different tools and programming languages out there, but where is comes to developer productivity, and pure fun of coding, there is nothing like RAD Studio and Delphi!

REFERENCES

- Demo Source Code: http://cc.embarcadero.com/item/30490
- JSON Components source code: https://github.com/pglowack/DelphiJSONComponents
- JSON Viewer source code: https://github.com/pglowack/DelphiJSONViewer
- JSON Viewer Application (Executable): http://cc.embarcadero.com/item/30495
- Webinar replay of "Learn How to Use the New JSON Features in RAD Studio 10 Seattle" https://www.youtube.com/watch?v=onX1MoE3mUM
- Blog Post: http://community.embarcadero.com/blogs/entry/learn-how-to-use-the-new-json-features-in-rad-studio-10-seattle-webinar-march-2nd-wednesday
- RAD Studio 10 "Seattle" Documentation http://docwiki.embarcadero.com/CodeExamples/Seattle/en/Main_Page
- RAD Studio Home Page: http://www.embarcadero.com/products/rad-studio
- C++Builder Home Page: http://www.embarcadero.com/products/cbuilder
- Delphi Home Page: http://www.embarcadero.com/products/delphi
- JSON Home Page: http://json.org
- JSON RFC document: https://www.ietf.org/rfc/rfc4627.txt

About The Author

Paweł Głowacki is Embarcadero's European Technical Lead for Developer Tools. Previously, Paweł spent over 7 years working as a senior consultant and trainer for Delphi within Borland Education Services and CodeGear. As well as working with Embarcadero customers across the region, he also represents Embarcadero internationally as a conference and seminar speaker. For more information check out Paweł's technical blog at http://community.embarcadero.com

Embarcadero Technologies, Inc. is the leading provider of software tools that empower application developers and data management professionals to design, build, and run applications and databases more efficiently in heterogeneous IT environments. Over 90 of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero's award-winning products to optimize costs, streamline compliance, and accelerate development and innovation. Founded in 1993, Embarcadero is headquartered in San Francisco with offices located around the world. Embarcadero is online at www.embarcadero.com.