

White Paper

REST in Delphi 2010

Marco Cantù, <http://blog.marcocantu.com>

November 2009

EXECUTIVE SUMMARY

Representational State Transfer (REST) is a new architecture for Web Services that is having a significant impact on the industry. Most of the new public web services from large vendors (Google, Yahoo, Amazon, now also Microsoft) rely on REST as the technology for sharing and merging information from multiple sources.

As implementing the REST architecture implies only using simple technologies (like HTTP and XML) Delphi has historically had good support for it. Now version 2010 adds specific support for the development of REST servers, part of the DataSnap infrastructure.

This paper delves in the technologies involved in REST from the Delphi perspective, showing both how to create client applications for popular Web sites and how to build REST servers using the specific Delphi 2010 support.

INTRODUCTION

Over the last ten years we have witnessed the explosion of the Web, and now of so-called Web 2.0. What we are only starting to see is the automatic interaction between different web sites, between web sites and client applications, between web sites and business databases, in a global interconnection that is often hard to fully understand.

On the Web, data is moving faster than we can browse it, so there is a strong demand for programs that can find, track and monitor information coming from diverse sources such as sales data, financial information, online communities, marketing campaigns, and many others.

WHY WEB SERVICES?

The rapidly emerging web services technology has the potential to change the way the Internet works for businesses. Browsing web pages to enter orders is fine for individuals (business-to-consumer [B2C] applications) but not for companies (business-to-business [B2B] applications). If you want to buy a few books, going to a book vendor website and punching in your requests is probably fine. But if you run a bookstore and want to place hundreds of orders a day, this is far from an efficient approach, particularly if you have a program that helps you track your sales and determine reorders. Grabbing the output of this program and reentering it into another application is ridiculous.

Web services are meant (or to be more precise were originally meant) to solve this issue: The program used to track sales can automatically create a request and send it to a web service, which can immediately return information about the order. The next step might be to ask for a tracking number for the shipment. At this point, your program can use another web service to track the shipment until it is at its destination, so you can tell your customers how long they have to wait. As the shipment arrives, your program can send a reminder via SMS or pager or Twitter to the people with pending orders, issue a payment with a bank web service, and...I could continue but I think I've given you the idea. Web services are meant for computer interoperability, much as the Web and e-mail let people interact.

The topic of web services is broad and involves many technologies and business-related standards. As usual, I'll focus on the underlying Delphi implementation and the technical side of web services, rather than discuss the larger picture and business implications. Delphi for Win32 offers some rather sophisticated support for web services, which originally came in the form of SOAP, and can now easily be extended by means of HTTP components and REST.

WEB SERVICE TECHNOLOGIES: SOAP AND REST

The idea of a web service is rather abstract. When it comes to technologies, there are currently two main solutions that are attracting developers. One is the use of the SOAP standard (Simple Object Access Protocol, see the reference site at <http://www.w3.org/TR/soap/>), another is the use of a REST (Representational State Transfer) approach, along with its variation XML-RPC (XML-Remote Procedure Call).

What is relevant to notice is that both solutions generally use HTTP as the transmission protocol (although they do provide alternatives) and use XML (or JSON) for moving the data back and forth. By using standard HTTP, a web server can handle the requests, and the related data packets can pass through firewalls.

In this white paper I won't provide details on SOAP, but focus exclusively on REST. I'll start by providing some theoretical foundations, show a simple "hand-made" example of a server and a client, delve into the development of REST clients for popular REST web services and focus on the REST server side support available in Delphi 2010 as an extension of the DataSnap architecture.

THE CONCEPTS BEHIND REPRESENTATIONAL STATE TRANSFER

Even if the general idea of REST has been around for some time, the introduction of this formal name and the theory behind it are fairly recent. What is relevant to mention up front is that there isn't a formal REST standard.

The term REST, an acronym for Representational State Transfer, was originally coined by Roy Fielding in his Ph.D. dissertation in year 2000, and spread very rapidly as synonymous with accessing data over the web using HTTP and URLs, rather than relying on the SOAP standard.

The term REST was originally used to describe an architectural style which described the relationship of a web browser with a server. The idea is that when you access a web resource (either using a browser or a specific client application) the server will send you a representation of the resource (an HTML page, an image, some raw data...). The client receiving the representation is set in a given state. As the client accesses further information or pages (maybe using a link) its state will change, transferring from the previous one.

In Roy Fielding's words:

"Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

REST ARCHITECTURE'S KEY POINTS

So, if REST is an architecture (or even better, an architectural style) it is clearly not a standard, although it uses several existing standards like HTTP, URL, plus many format types for the actual data.

While SOAP relies on HTTP and XML but builds on those, REST architectures use HTTP and XML (or other formats) exactly as they are:

- REST uses URLs to identify a resource on a server (while SOAP uses a single URL for many requests, detailed in the SOAP envelope). Notice the idea is to use the URL to identify a resource not an operation on the resource.
- REST uses HTTP methods to indicate which operation to perform (retrieve or HTTP GET, create or HTTP PUT, update or HTTP POST, and delete or HTTP DELETE)
- REST uses HTTP parameters (both as query parameters and POST parameters) to provide further information to the server
- REST relies on HTTP for authentication, encryption, security (using HTTPS)
- REST returns data as plain documents, using multiple mime formats (XML, JSON, images, and many others)

There are quite a few architectural elements that are worth considering in this kind of scenario. REST demands for system to be:

- Client/server in nature (nothing directly to do with database RDBMS here)
- Inherently stateless
- Cache-friendly (the same URL should return the same data if called twice in sequence, unless the server side data changed), permitting proxy and cache servers to be inserted between the client and the server. A corollary is that all GET operations should have no side effect

There is certainly much more to the theory of REST than this short section covered, but I hope this got you started with the theory. The practical examples coming next along with Delphi code should clarify the main concepts.

REST TECHNOLOGIES AND DELPHI

Having said that, there is no REST standard and you do not need to use specific tools for REST development, there are existing standards that REST relies upon and that is worth introducing

shortly (an in-depth description of each could as well take an entire book). The specific focus here is also Delphi support for these technologies.

HTTP (CLIENT AND SERVER)

The HyperText Transfer Protocol is the standard at the heart of the World Wide Web, and needs no introduction. Granted, HTTP can be used by Web Browsers, but also by any other application.

In Delphi applications the simplest way to write a client application that uses HTTP is to rely on the Indy HTTP client component, or `IdHttp`. If you call the `Get` method of this component, providing a URL as parameter, you can retrieve the content of any Web page and many REST servers. At times, you might need to set other properties, providing authentication information or attach a second component for SSL support (as we'll see in some examples). The component supports the various HTTP methods, beside `Get`.

Notice that to be on the safe side you should generally make `IdHttp` requests inside a thread, as the Indy suite uses blocking threads: the user interface of your program will be stuck until the requests are returned (which take a long time in case of a slow web server or a large data transfer). I won't generally use threads in the demos in this paper only for the sake of simplicity, but that is the recommended approach.

On the server side you can use multiple architectures for creating a web server or web server extension in Delphi. For a stand-alone web server you can use the `IdHttpServer` component, while for creating web server extensions (CGI applications, ISAPI, or Apache modules) you can use the WebBroker framework. Another new option is given by the HTTP support of `DataSnap` in Delphi 2010.

XML

Extended Markup Language is a commonly used format for data, although many REST servers also use alternative data structures like JSON (JavaScript Object Notation) and at times even plain comma-delimited text files. Again, XML is very popular and I don't want to cover it in detail here.

In Delphi, you can process XML documents using the `XmlDocument` component. This is a wrapper around one of the available XML DOM engines (the default one being Microsoft XML DOM). Once a document is loaded you can navigate its node structure or query the document using XPath (which is often the style I prefer).

XPATH

XPath is a query language that lets you identify and process the nodes of an XML document. The notation used by XPath resembles file system paths (`/root/node1/node2`) with square brackets added to express conditions on node attributes or subnodes (`root/node1[@val=5]`) or even complex expressions. The result of an XPath statement can be in itself an expression, like the number of nodes matching a rule or the total value of a set of nodes.

In Delphi you can execute an XPath request by applying it to the DOM hosting the document, in case the specific DOM supports it. We'll see an example of XPath in the first demo below.

REST CLIENTS WRITTEN IN DELPHI

There are countless examples of REST servers that you can find on the Web, from the classic Amazon Web Service, now rebranded as "Amazon E-Commerce Service" since the Amazon Web Service name is used for their Cloud Computing offering, to many sites letting you have access to information using a XML data structure rather than the HTML format.

Even if the number of web services that uses REST on the Internet is high, most actual web services require some developer token (as covered in some of the coming demos), while only a handful offers free and open access. For a slightly different list of Delphi REST clients and for the source code of these demos you can refer to the specific section of one of my web sites:

<http://ajax.marcocantu.com/delphirest/default.htm>

A REST CLIENT FOR RSS FEEDS

The most widespread format for distributing information as XML is the use of the RSS and ATOM feeds, mostly attached to blog and news sites, but equally usable for any data source. The interesting point about feeds is they provide to client applications the same information a user will generally access using a web browser. Feed information is processed by these client applications, and at times even combined in a summary of similar feeds, as happens on the Delphi Feeds site.

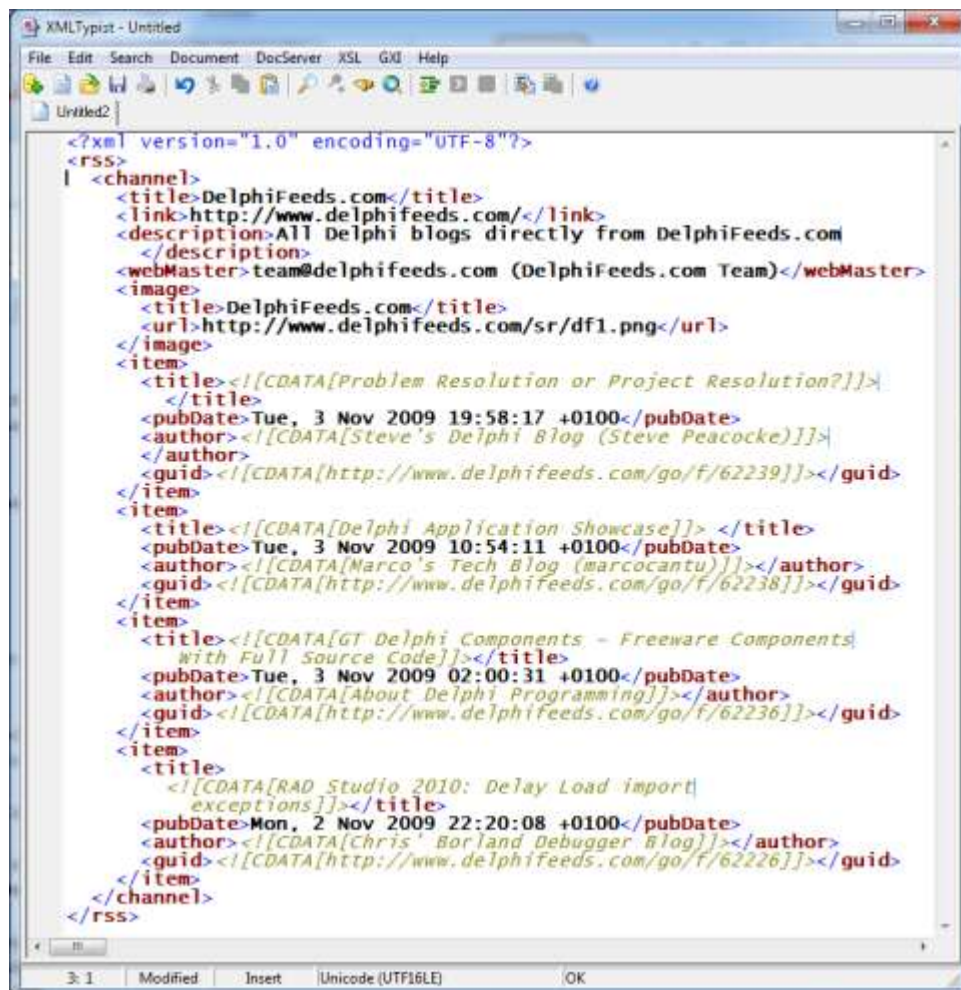
That's why, as a first example of a client application using REST, I wrote a very simple RSS client looking into Delphi Blogs at <http://www.delphifeeds.com>. Every time you access dynamic XML data using an URL and you can change the URL to access different data, you are using the REST approach.

THE REST CALL AND THE XML DOCUMENT

The RssClient program uses an IdHttp component and an XMLDocument component. The first is used to grab the data from the Web (that is, making the REST client call) and loads the data in the latter component:

```
var
    strXml: string;
begin
    strXml := IdHTTP1.Get ('http://feeds.delphifeeds.com/delphifeeds');
    XMLDocument1.LoadFromXML(strXml);
```

The data extracted would look like the following (which I've somewhat simplified for readability), when displayed in an XML editor:



PROCESSING THE RSS DATA WITH XPATH

To extract the relevant information from this XML document the RssClient program uses XPath expressions. For example to read the title of the first blog post (item) of the list it uses:

```
/rss/channel/item[1]/title
```

This is done in a cycle along with the extraction of some other information, formatted and displayed in a list box. Using XPath requires the use of a custom interface of the Microsoft engine: hence the cast to the extended interface IDOMNodeSelect.

Once it has the nodes it is interested in, the program looks for any child text node, using a getChildNodes helper function I wrote for this purpose, and adds the data to a listbox. This is the complete code of the method executed when the Update button of the program is pressed:

```

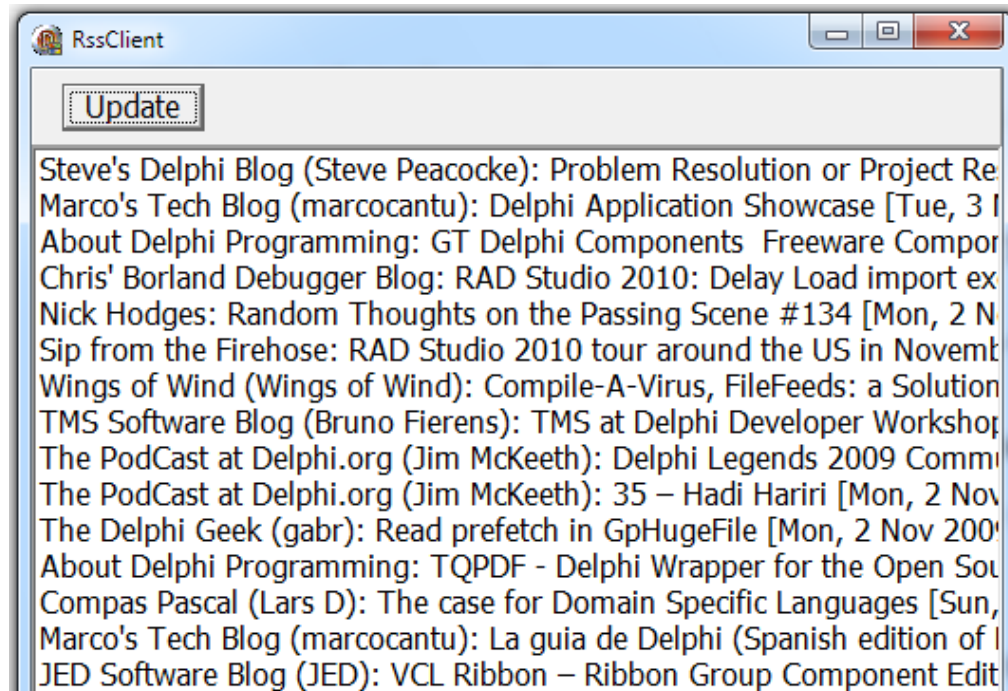
procedure TRssForm.btnUpdateClick(Sender: TObject);
var
    strXml, title, author, pubDate: string;
    I: Integer;
    IDomSel: IDOMNodeSelect;
    Node: IDOMNode;
begin
    strXml := IdHTTP1.Get ( 'http://feeds.delphifeeds.com/delphifeeds' );

    XMLDocument1.LoadFromXML(strXml);
    XMLDocument1.Active := True;
    IDomSel := (XMLDocument1.DocumentElement.DOMNode
        as IDOMNodeSelect);

    for I := 1 to 15 do
    begin
        Node := IDomSel.selectNode(
            '/rss/channel/item[' + IntToStr (i) + ']/title');
        title := getChildNodes (Node);
        Node := IDomSel.selectNode(
            '/rss/channel/item[' + IntToStr (i) + ']/author');
        author := getChildNodes (Node);
        Node := IDomSel.selectNode(
            '/rss/channel/item[' + IntToStr (i) + ']/pubDate');
        pubDate := getChildNodes (Node);
        ListBox1.Items.Add(author + ': ' + title + ' [' + pubDate + ']');
    end;
end;

```

The effect of running this program is visible in the screenshot below:



OF MAPS AND LOCATIONS

Access to location and map information can be very useful in multiple circumstances, as many applications have to do with addresses. In the recent years, more and more mapping data has been made available on the web by multiple large sites, including Google, Yahoo, and Microsoft.

GOOGLE GEOCODING SERVICE

The first service of this category I'm going to use is Google's Geocoding service. This service lets you submit an address and retrieve its latitude and longitude, with requests like:

`http://maps.google.com/maps/geo?q=[address]&output=[format] &key=[key]`

You can type a similar URL also in your browser for testing purposes, as you can see in the picture in the next page showing New York coordinates in a browser (using the XML format):



The GeoLocation example I've built uses the addresses the companies of the classic Customer.cds sample database that comes with Delphi (it uses a local copy, in the ZIP file along with the project source code).

As with many similar services, this is free for limited usage (the program has some extra sleep() calls to avoid hitting the maximum rate per minute), but requires a registration for the specific service on <http://code.google.com>.

The demo program requires your *devkey* to be added to a *GeoLocation.ini* file which must reside in the user's document folder and has the simple structure:

```
[googlemap]
devkey=
```

RESOLVING CUSTOMER ADDRESSES

The program works in two steps.

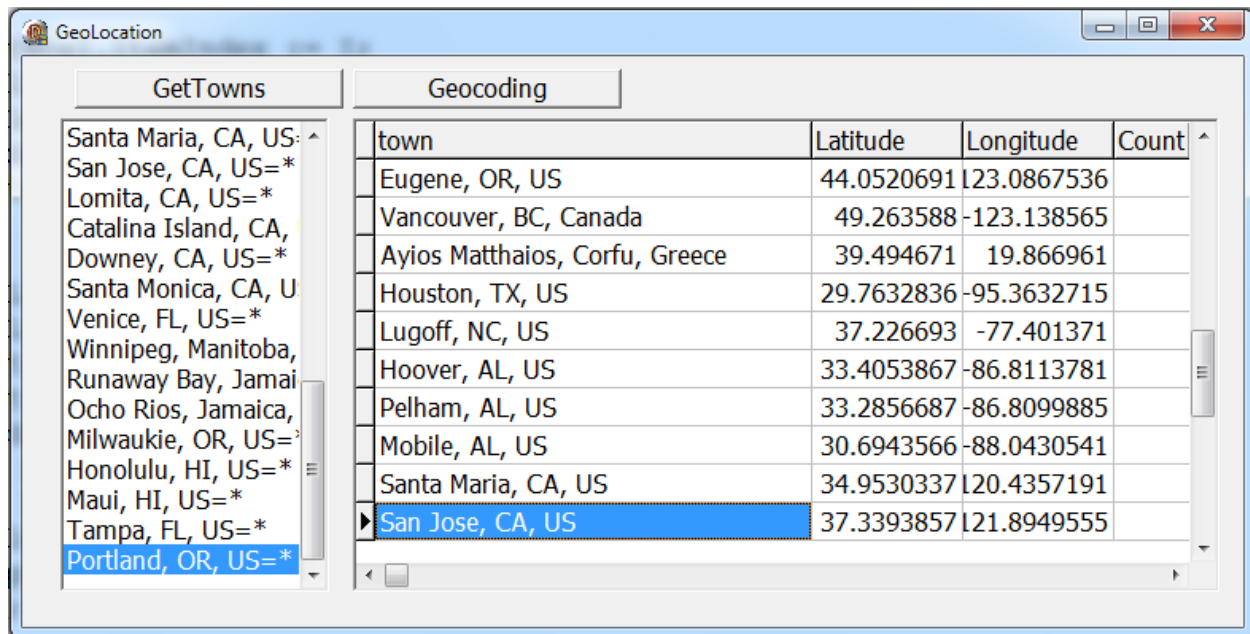
- First, it looks for unique names of cities / state / country, by scanning the ClientDataSet component and filling a string list. This code is unrelated to REST, so I've omitted it from this paper.
- The second step is to look up each city on the Google Geocoding service, filling an in-memory ClientDataSet with the resulting information.

This time rather than asking for the XML version of the data, I resorted to a simpler CSV format, which the program parses using a StringList object.

Here is the actual *Geocoding* code:

```
procedure TFormMap.btnGeocodingClick(Sender: TObject);  
var  
    I: Integer;  
    strResponse, str1, str2: string;  
    sList:TStringList;  
begin  
    cdsTown.Active := False;  
    cdsTown.CreateDataSet;  
    cdsTown.Active := True;  
    sList := TStringList.Create;  
  
    for I := 0 to sListCity.Count - 1 do  
    begin  
        ListBox1.ItemIndex := I;  
        if Length (sListCity.Names[I]) > 2 then  
        begin  
            strResponse := IdHTTP1.Get( TIDUri.UrlEncode(  
                'http://maps.google.com/maps/geo?q=' +  
                (sListCity.Names[I]) + '&output=csv&key=' +  
                googleMapKey));  
  
            sList.LineBreak := ',';  
            sList.Text := strResponse;  
            str1 := sList[2];  
            str2 := sList[3];  
            cdsTown.AppendRecord([sListCity.Names[I],  
                StrToFloat (str1), StrToFloat (str2),  
                Length (sListCity.ValueFromIndex[I])]);  
  
            Sleep (150);  
            Application.ProcessMessages;  
        end;  
    end;  
    sList.Free;  
end;
```

The output should look like the image in the following page:



YAHOO MAPS

As a further step, we can try to access to the actual map corresponding to an address. If Google Maps provide countless features, they are meant to be hosted on web sites not on client applications (although I do have an example of hosting a Google Map in a client program, its architecture and code are quite complex and won't fit in this paper).

The new example, called YahooMaps uses Yahoo Map API to get an actual map and show it in an Image control. Information about this REST API and the link to obtain a free Yahoo Application ID are available at:

<http://developer.yahoo.com/maps/>

Again, to run the programs you'll have to obtain this ID and store in a specific INI file in the "user documents" folder called YahooMaps.ini.

The map is retrieved in two steps: a first HTTP call passes the address and receives the URL of the map image, which is retrieved using a second HTTP call. Again, you could simulate the two steps in a web browser, which is quite nice for debugging purposes.

While the program uses the same database and intermediate StringList of the previous example, it also has a button that it uses to display the map or a hard-coded city (San Jose, California), using the following method:

const

```

    BaseUrl = 'http://api.local.yahoo.com/MapsService/V1/';

procedure TFormMap.Button1Click(Sender: TObject);
var
    strResult: string;
    memStr: TFileStream;
begin
    strResult := IdHTTP1.Get(BaseUrl +
        'mapImage?' +
        'appid=' + yahooAppid +
        '&city=SanJose,California');

    XMLDocument1.Active := False;
    XMLDocument1.XML.Text := strResult;
    XMLDocument1.Active := True;
    strResult := XMLDocument1.DocumentElement.NodeValue;
    XMLDocument1.Active := False;

    // now let's get the referred image
    memStr:= TFileStream.Create ('temp.png', fmCreate);
    IdHttp1.Get(strResult, memStr);
    memStr.Free;

    // load the image
    Image1.Picture.LoadFromFile('temp.png');
end;

```

The first HTTP Get request provides the actual query and returns an XML document with the URL of the image of the actual map, which looks like (with long IDs omitted):

```

<Result>
  http://gws.maps.yahoo.com/mapimage?MAPDATA=[...]&mvt=m
  &cltype=onnetwork&.intl=us&appid=[...]
  &oper=&_proxy=ydn,xml
</Result>

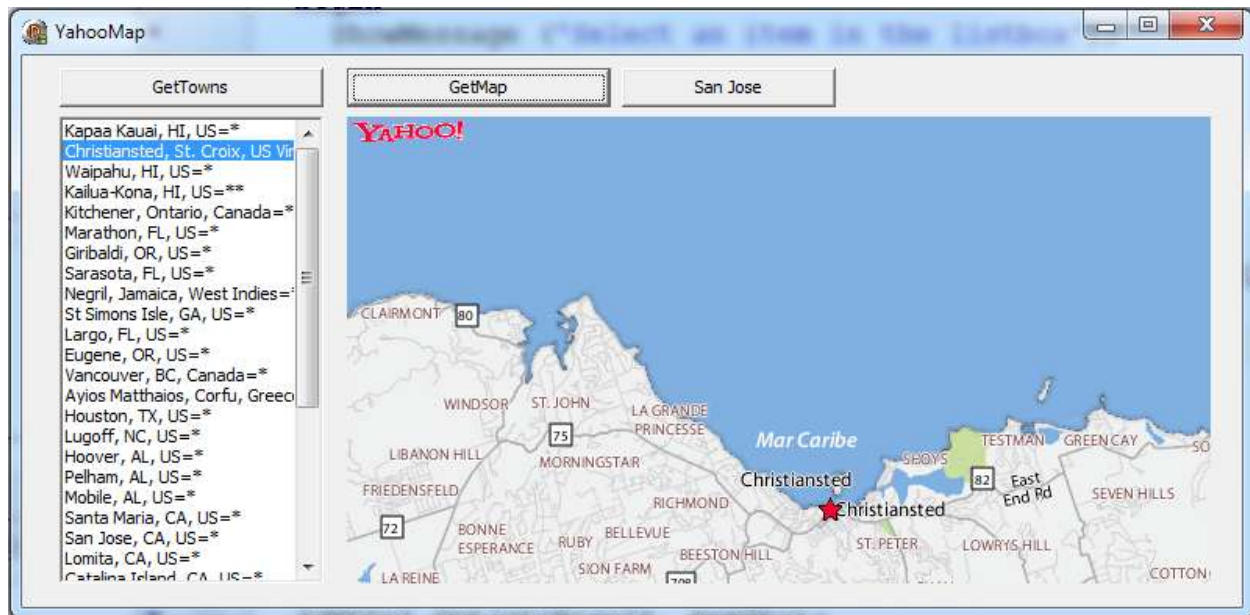
```

That's why the program can extract the value of the only node with the code:

```
XMLDocument1.DocumentElement.NodeValue
```

Finally, the image is saved to a temp file and loaded into an Image control. Beside the map of this specific city, the program can also fetch those of the Customer.cds database of the previous example. There is a button to get the map of a hardcoded location (San Jose, as this was the Delphi Live demo).

Here is an example:



GOOGLE TRANSLATE API

Another simple and interesting example of a REST API provided by Google is their translation service, called Google Translate REST API. The documentation is at:

<http://code.google.com/apis/ajaxlanguage/documentation/>

In this case there is no need for a signup key (and an INI file), but only provide a referrer site (although everything seems to work even without that information). You can ask for a translation in your Web Browser by entering an URL like:

<http://ajax.googleapis.com/ajax/services/language/translate?v=1.0&q=what%20a%20nice%20day&langpair=en|de>

The output of this call is visible below (I have also listed the JSON result for readability):



```
{
  "responseData":
```

```

{
  "translatedText": "Was für ein schöner Tag"
},
"responseDetails": null,
"responseStatus": 200
}

```

This example takes one step further compared to previous demos. Rather than making the HTTP request, it uses a specific custom VCL component, invoked with a class method (so you don't need to place the component on a form, even if you could). This support component makes the API very easy to use, and encapsulates the HTTP call completely.

A TRANSLATION COMPONENT

This is the declaration of the class of the component:

```

type
  TBabelGoogleRest = class (TComponent)
  protected
    Http1: TIdHttp;
    FFromLang: string;
    FToLang: string;
    FActiveInForm: Boolean;
    procedure SetFromLang(const value: string);
    procedure SetToLang(const value: string);
  public
    function DoTranslate (strIn: string): string; virtual;
    constructor Create(AOwner: TComponent); override;

    class function Translate (strIn, langIn, langOut: string): string;
  published
    property FromLang: string read FFromLang write SetFromLang;
    property ToLang: string read FToLang write SetToLang;
  end;

```

The actual processing is in the DoTranslate function:

```

function TBabelGoogleRest.DoTranslate(strIn: string): string;
var
  strUrl, strResult: string;
  nPosA, nPosB: Integer;
begin
  strUrl := Format (
    'http://ajax.googleapis.com/ajax/services/language/translate?' +
    'v=1.0&q=%s&langpair=%s',

```

```

    [TidUri.ParamsEncode (strIn),
    FFromLang + '%7C' + FToLang]); // format hates the %7 !!!
strResult := Http1.Get(strUrl);

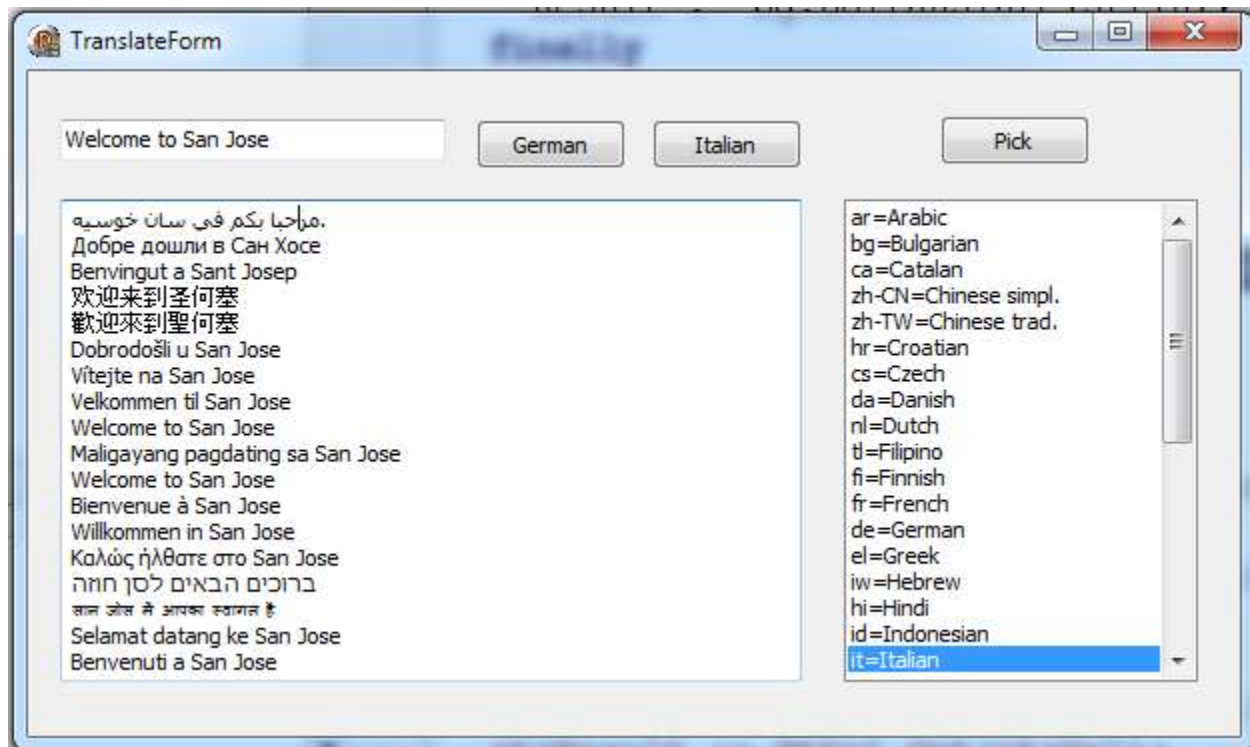
nPosA := Pos ('"translatedText":', strResult); // begin of JSON data
if nPosA = 0 then
begin
    nPosA := Pos ('"responseDetails":', strResult);
    nPosA := nPosA + Length ('"responseDetails":');
end
else
    nPosA := nPosA + Length ('"translatedText":');

nPosA := PosEx ('"', strResult, nPosA) + 1; // opening quotes
nPosB := PosEx ('"', strResult, nPosA) - 1; // end quotes
Result := Copy (strResult, nPosA, nPosB - nPosA + 1);
end;

```

The result of the request to the given URL is in JSON format (as this is considered as a Javascript API by Google). I've decided to process the JSON information manually, although I could have used the new JSON to objects mapping support in Delphi 2010, which I'll discuss later. The actual method can be called using a class method that creates a temporary object, sets its properties, and calls the DoTranslate function. There are some other scenarios, but the remaining code should be easy to understand.

The main form of the demo program has a list box filled with all supported languages. The demo translates from English, but you can set it up on the opposite direction as well. In theory, any two language token combination works, in practice not always. Once you ask for a translation, the result is added to a log. Here I've applied the call to the first group of languages (in alphabetic order):



TWITTER, AS SIMPLE AS IT COULD BE

Twitter web services interface has significantly contributed to the rise of this simple social web site, as it has spawned an entire ecosystem of twitter-enabled applications. As simple as Twitter is for users, so is its REST web service interface. Twitter was notably written in Ruby, and Ruby on Rails has certainly contributed pushing the idea of REST, with its neat mapping of URLs to internal application resources, of the foundations of the REST approach.

If you have a Twitter account how do you access your last 20 entries? And how do you access the last 20 entries on Twitter? These are the URLs:

```
http://twitter.com/statuses/user_timeline.xml
http://twitter.com/statuses/public_timeline.xml
```

The only condition is to make a GET HTTP request passing the standard HTTP headers with the user name and the password, something you can easily add in Delphi to the corresponding properties of an `IdHttp` component. A much more complicated operation, if we can say so, is posting a status update to your account. This is achieved with a POST HTTP request that passes a status parameter, as in the following code snippet, which send Twitter the text of one of the fields of the current record of a `ClientDataSet` component (`ClientDataSet1TEXT`), marking it as posted:

```

procedure TForm34.btnPostCurrentClick(Sender: TObject);
var
    strResult: string;
    listParams: TStringList;
begin
    listParams := TStringList.Create;
    listParams.Add('status=' + ClientDataSet1TEXT.AsString);
    try
        strResult := idhttp1.Post(
            'http://twitter.com/statuses/update.xml', listParams);
        ShowMessage(strResult);
        ClientDataSet1.Edit;
        ClientDataSet1POSTED.AsString := 'T';
        ClientDataSet1.Post;
    finally
        listParams.Free;
    end;
end;

```

This code is extracted from my *"Delphi Tweet of the Day"* application, which uses a ClientDataSet (local or hooked with a remote server) for entering the tweets. After posting the value of the TEXT field of the table, in fact, the program sets the POSTED field to True.

There is some other code in the program, but nothing really related with Twitter itself or its REST API. The other element relevant for our discussion is the configuration of the IdHttp component:

```

object IdHTTP1: TIdHTTP
    Request.ContentLength = -1
    Request.Accept = 'text/html, */*'
    Request.BasicAuthentication = True
    Request.Password = '***' // omitted but required
    Request.UserAgent = 'tweetoftheday'
    Request.Username = 'delphitweetday'
    HTTPOptions = [hoForceEncodeParams]
end

```

In other words coding to Twitter is extremely simple and you can benefit from automatic posting at fixed time intervals or when something relevant happens to your business (and in your database).

INTERFACING WITH GOOGLE SPREADSHEET SERVICES

If working with map and translations can be a lot of fun, and posting to Twitter can become part of your company marketing activities, I think it will become increasingly relevant over time to be able to interact with "business" services like Google Docs:

<http://docs.google.com/>

The Web Services interface for Google Docs lets you upload and download (and even convert) web documents and manage their access credentials, which is certainly quite nice. But, there is also one interface that lets you work with the content of a given document on the web: this is the Google Spreadsheet Services API, still in an early beta at the time of writing.

Using this API you can access individual cells of any sheet of any spreadsheet that is either private or public on the system. Anyone looking at those documents with a web interface will see them change in real time!

As this API (like many other Google APIs) allows access to personal and reserved documents, it requires a more robust security layer than other REST APIs we have used so far. There are two notable differences compared to simpler solutions:

- The API uses HTTPS rather than HTTP. You'll need to hook OpenSSL or another SSL library to the client application and call the proper Indy support code.
- The API requires a separate authentication request, which will return an authorization *token* that will expire after some time. You'll have to pass this token into any following request (coming from the same IP address).

Let's look at this infrastructure before delving into the specific problem (by the way, I wrote this authorization management classes for a different Google Web Services, the Provisioning API, used to set up accounts on private and paid Google Mail and Docs domains).

To support authentication and make a request using this support, I've written a specific Delphi class, TGoogleAuth, which has the following public interface (I've removed the many private fields and property access methods):

```
type
  TGoogleAuth = class
  public
    property GoogleEmail: string
      read FGoogleEmail write SetGoogleEmail;
    property GooglePwd: string
      read FGooglePwd write SetGooglePwd;
    property AuthString: string
      read GetAuthString write SetAuthString;
    property ReqTime: TDateTime
```

```

        read FReqTime write SetReqTime;
    property AccountType: string
        read FAccountType write SetAccountType;
    property ServiceName: string
        read FServiceName write SetServiceName;
public
    function Expired: Boolean;
    procedure Renew;
end;

```

Four of the properties above (Email, password, type of account and name of service) are input values, while the other two are the authorization string and the time it was set, which is used to automatically renew it again a given amount of times. As you ask for this token, the class will check if it needs to get a new one. This is the getter method of the AuthString property:

```

function TGoogleAuth.GetAuthString: string;
begin
    if FAuthString = '' then
        Renew;
    if Expired then
        Renew;
    Result := FAuthString;
end;

```

The authorization request code is inside the Renew method, which uses an SSL HTTP connection to ask for the authorization token passing the user name and password in an encrypted format:

```

procedure TGoogleAuth.Renew;
var
    res, req: String;
    sList: TStringList;
    IdHttp: TIdHttp;
begin
    FAuthString := '';

    IdHttp := TIdHttp.Create (nil);
    try
        IdHttp.IOHandler := TIdSSLIOHandlerSocketOpenSSL.Create(IdHttp);
        req := 'https://www.google.com/accounts/ClientLogin?Email=' +
            FGoogleEmail + '&Passwd=' + FGooglePwd +
            '&accountType=' + FAccountType + '&service=' + FServiceName;
        res := IdHttp.Get (req);
    finally
        idHttp.Free;
    end;
end;

```

```

sList := TStringList.Create;
try
  sList.Text := res;
  FAuthString := sList.Values[ 'Auth' ];
  FReqTime := Now;
finally
  sList.Free;
end;
end;
end;

```

I created a global singleton for this class, and every time I need to make a request I pass through a helper method (a global function), which adds the extra token. This is the code of this rather long function:

```

function DoRequest (
  const methodAttr, fromAttr, strData: string): string;
var
  res: String;
  postStream: TStream;
  IdHttp: TIdHttp;
  resStream: TStringStream;
begin
  IdHttp := TIdHttp.Create (nil);
  try
    // add authorization from stored key
    IdHttp.Request.CustomHeaders.Values [ 'Authorization' ] :=
      'GoogleLogin auth=' + googleAuth.AuthString;
    IdHttp.Request.CustomHeaders.Values [ 'Content-type' ] :=
      'application/atom+xml';
    IdHttp.Request.CustomHeaders.Values [ 'GData-Version' ] := '2';

    // use SSL
    IdHttp.IOHandler := TIdSSLIOHandlerSocketOpenSSL.Create(IdHttp);
    try
      if (methodAttr = 'post') or (methodAttr = 'put') then
        begin
          postStream := TStringStream.Create (strData);
          try
            postStream.Position := 0;
            if (methodAttr = 'post') then
              res := IdHttp.Post (fromAttr, postStream)
            else // (methodAttr = 'put')
              res := IdHttp.Put (fromAttr, postStream);
          finally

```

```

        PostStream.Free;
    end;
end
else if (methodAttr = 'delete') then
begin
    resStream := TStringStream.Create ('');
    try
        IdHttp.DoRequest (hmDelete, fromAttr, nil, resStream);
    finally
        resStream.Position := 0;
        res := resStream.DataString;
        resStream.Free;
    end;
end
else // 'get' or not assigned
    res := IdHttp.Get (fromAttr);
except
    on E: Exception do // intercept and use as result (which is odd)
    begin
        res := E.Message;
    end;
end;
finally
    IdHttp.Free;
end;
Result := res;
end;

```

This was a lot of structural code, but it was worth it to make our SSL-based and authorized calls easy. As an example, the code I'll have to use to request a list of private spreadsheets becomes the following:

```

DoAppRequest (
    'get',
    'http://spreadsheets.google.com/feeds/spreadsheets/private/full',
    '');

```

This code is part of the actual demo, which uses the information available in the dbtosheet.ini file, which has the following structure and is required to run the demo program:

```

[google]
email=
password=
accounttype=GOOGLE

```

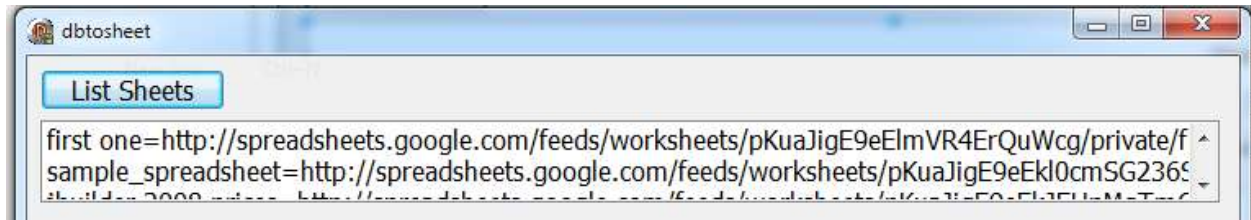
The file is loaded when the program starts and the three values are used to fill in the global singleton TGoogleAuth object, named googleAuth. The fourth parameter, the service, is set up by the program:

```
googleAuth.GoogleEmail :=  
  inifile.ReadString('google', 'email', '');  
googleAuth.GooglePwd :=  
  inifile.ReadString('google', 'password', '');  
googleAuth.AccountType :=  
  inifile.ReadString('google', 'accounttype', 'GOOGLE');  
googleAuth.ServiceName := 'wise';
```

With this configuration in place, the program has a first button that makes the request above for the list of available spreadsheets and adds their identifiers (or URLs) in a list box. The resulting XML is parsed using XPath, but only after stripping the namespace, or else the XPath requests should take the namespace into account and become very complex:

```
procedure TForm34.btnListSheetsClick(Sender: TObject);  
var  
  strXml: string;  
  IDomSel: IDOMNodeSelect;  
  Node: IDOMNode;  
  I, nCount: Integer;  
  title, id: string;  
begin  
  strXml := DoAppRequest ('get',  
    'http://spreadsheets.google.com/feeds/spreadsheets/private/full',  
    '');  
  strXml := StringReplace (strXml,  
    '<feed xmlns='http://www.w3.org/2005/Atom'', '<feed ', []);  
  
  XMLDocument1.LoadFromXML(strXml);  
  XMLDocument1.Active := True;  
  
  IDomSel := (XMLDocument1.DocumentElement.DOMNode as IDOMNodeSelect);  
  nCount := IDomSel.selectNodes ('/feed/entry').length;  
  for I := 1 to nCount do  
  begin  
    Node := IDomSel.selectNode(  
      '/feed/entry[' + IntToStr (i) + ']/title');  
    title := getChildNodes (Node);  
    Node := IDomSel.selectNode(  
      '/feed/entry[' + IntToStr (i) + ']/content/@src');  
    id := getChildNodes (Node);  
    ListBox1.Items.Add(title + '=' + id);  
  end;  
end;
```

The for loop works on each node (it uses the count XPath function to determine the number of nodes), adding the spreadsheet title and its ID/URL to the list box, as follows:



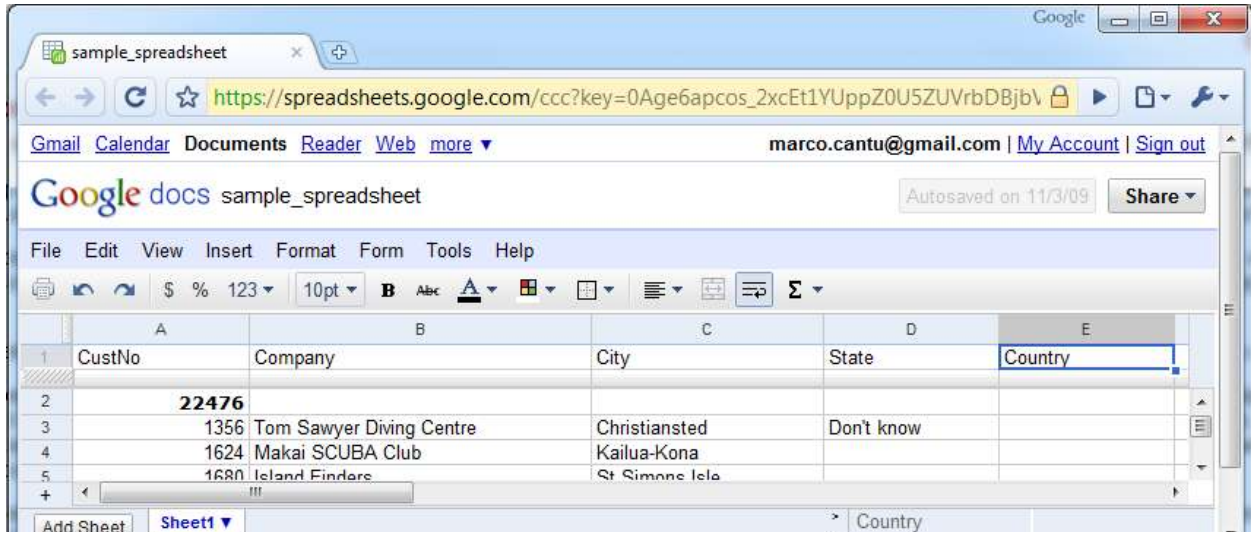
Now given the ID/URL you can make a second query to ask for the available tab sheets inside the given document. Even if there is only one, you'll need to refer to a specific tab in each operation affecting the document content. This is how the program retrieves the ID from the list box and makes the second REST request:

```
strSheetId := ListBox1.Items.ValueFromIndex [ListBox1.ItemIndex];  
strXml := DoAppRequest ('get', strSheetId, '');
```

At this point there is another similar loop to extract sheet names and move them to a second list box:

```
IDomSel := (XMLDocument1.DocumentElement.DOMNode as IDOMNodeSelect);  
nCount := IDomSel.selectNodes ('/feed/entry').length;  
for I := 1 to nCount do  
begin  
    Node := IDomSel.selectNode(  
        '/feed/entry[' + IntToStr (i) + ']/title');  
    title := getChildNodes (Node);  
    Node := IDomSel.selectNode(  
        '/feed/entry[' + IntToStr (i) + ']/content/@src');  
    id := getChildNodes (Node);  
    ListBox2.Items.Add(title + '=' + id);  
end;
```

Now the real power of this program comes from the ability to add data to an actual document. The simplest way to accomplish this is to set up a document which behaves like a database table (that is, it has a first line with the field names). This is a document of this kind inside the browser:



The cells of the first row are simple text entries, nothing special. What is nice is that you can refer to the corresponding columns using a dynamic namespace like:

```
<gsx:Company>Tom Sawyer Diving Centre</gsx:Company>
```

This means we can add new rows to the document by fetching the fields of a table with corresponding names. This requires a POST HTTP request using the sheet ID:

```
const
  gsxNamespace =
    'xmlns:gsx="http://schemas.google.com/' +
    'spreadsheets/2006/extended"';

begin
  strSheetId := ListBox2.Items.ValueFromIndex [ListBox2.ItemIndex];

  Memo1.Lines.Add(
    DoAppRequest ( 'post',
      strSheetId,
      '<entry xmlns="http://www.w3.org/2005/Atom" ' +
        gsxNamespace + '>' +
        recordtoxml +
      '</entry>' ));
```

The recordtoxml function grabs the values of the fields we are interested in from the current record of the ClientDataSet used by the program, producing the proper input data:

```
function recordtoxml: string;
begin
  Result :=
    FieldToXml ( 'custno' ) +
```

```

    FieldToXml ('company') +
    FieldToXml ('city') +
    FieldToXml ('state') +
    FieldToXml ('country');
end;

function FieldToXml (fieldname: string): string;
begin
    Result := '<gsx:' + fieldname + '>' +
        ClientDataSet1.FieldByName(fieldname).AsString +
        '</gsx:' + fieldname + '>';
end;

```

As already mentioned, the *gsx:* pseudo namespace refers to the names of the spreadsheet columns, which are determined by the strings in its first line. This code lets you populate the spreadsheet with new lines corresponding to database records.

Now this doesn't prevent you (and anyone else with editing rights on the document) from editing the data once posted, it lets the document automatically recalculate totals based on the data sent by the client application, and makes it possible for anyone in the world with proper rights to see the resulting document.

So we have a sophisticated mechanism for viewing and editing data originating from database tables, automatically updated over time, with a robust permissions infrastructure... simply thanks to a free Gmail account!

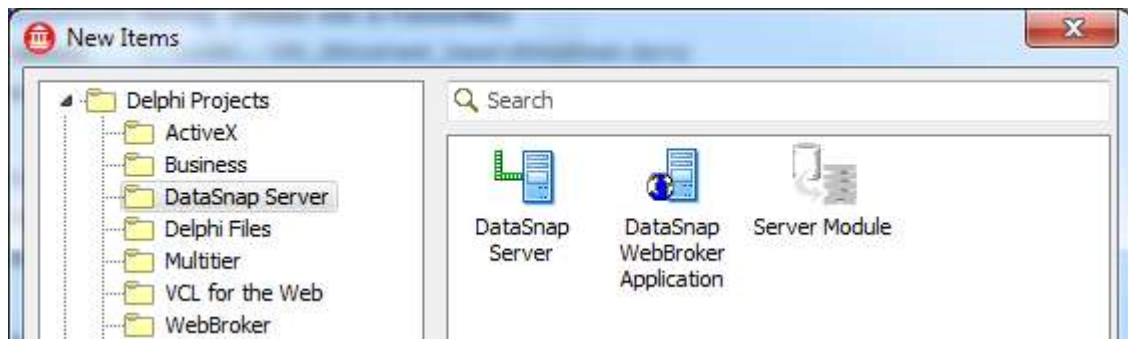
REST SERVERS IN DELPHI 2010

Now that we have spent a considerable amount of time looking at a diverse array of client REST applications written in Delphi, facing different permission requests and using different data type formats, we are ready to start delving into the second part of this white paper, focused on writing REST servers in Delphi 2010.

Don't get me wrong about this: you can write REST servers in any version of Delphi capable of creating a Web server, that is, probably since Delphi 3. Today you can use the *IdHTTPServer* component or the *WebBroker* architecture or other third-part solutions to roll out your own custom REST server. I did it over the years, as examples in my *Mastering Delphi 2005* book can prove. So here I'm deliberately focusing on one of the options, because it is well integrated into Delphi 2010 and will probably be the foundation of any future extension provided by Embarcadero Technologies to the product.

BUILDING A FIRST SIMPLE REST SERVER

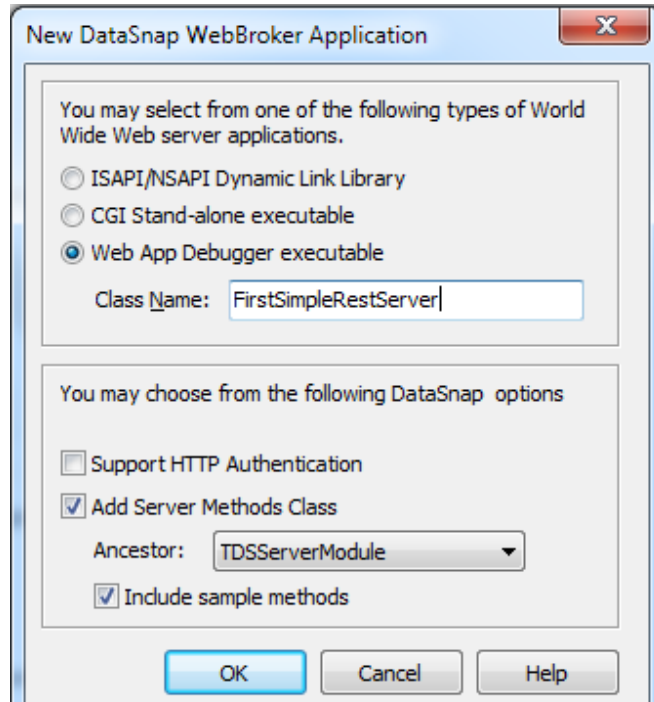
To build a first simple REST server in Delphi 2010 we can use the *DataSnap Wizard*:



As there are actually two DataSnap Wizards (as you can see above) we need to decide which one to pick. If you want to host your REST server as a web server, picking the DataSnap WebBroker Application will probably be your best choice, but you can later add components to the project generated by both wizards to make them very similar. In general, using a plain DataSnap server should make more sense if you plan having Delphi clients or other client applications and your server is on your internal network, while a DataSnap WebBroker server which integrates with your HTTP server, makes more sense for open architectures and for calling the REST server from Browser-based applications.

From a slightly different perspective, the DataSnap WebBroker server lets you have more control on the HTTP requests coming in and lets you integrate your REST data within your WebBroker server.

Having picked this option, you'll see a dialog box with a few settings. **The most important** decision you have to take in this Wizard is the type of application skeleton. Don't get too worried, though, as you'll be able to change the project architecture preserving all of your actual code later on:



For actual hosting in a web server, the ISAPI option is generally the recommended one, although the (generally slower) CGI is easier to set up and debug. But for true debugging and testing I really recommend using the Web App Debugger infrastructure provided by Delphi, which lets you add breakpoint to your code directly and even inspect the HTTP data flow, including the headers. If you pick that option insert an internal class name, which is used only as a reference to the program (in the Web App Debugger URL). Notice also that even if this is not explicit in the Wizard dialog box, Delphi still offers support for Apache server integration modules, although you'll have to set up this kind of project manually.

In the bottom portion of the DataSnap WebBroker Wizard dialog you can ask for a ready-to-use server method class with sample methods, but again it would be quite simple to roll out your own in code. With the settings above, Delphi will generate a project with three units:

- A main form, which has no specific use but can be used for logging information. This form will be removed as you change the project structure to a web server module or CGI, so don't bother much about it.
- A web data module (inheriting from TWebModule) hosting the DataSnap server components
- A data module acting as server class, where you add the code you want your REST server to execute.

THE WEB DATA MODULE GENERATED BY THE DATASNAP WEBBROKER WIZARD

Let's look at these two data module with some more detail, before we add some code to the program and test it. The web module is the core element of the WebBroker architecture. It can define multiple actions and has pre-processing and post-processing events for any HTTP request. On this web module you can add components which intercept given URL actions, as in this example the DSHTTPWebDispatcher:

```
object DSHTTPWebDispatcher1: TDSHTTPWebDispatcher
    RESTContext = 'rest'
    Server = DSServer1
    DSHostname = 'localhost'
    DSPort = 211
    webDispatch.MethodType = mtAny
    webDispatch.PathInfo = 'datasnap*'
end
```

This component intercepts any request with a URL starting with *'datasnap'*, which are passed to the HTTP support of DataSnap. For requests starting with *'datasnap'* and indicating a *'rest'* path, the processing will be diverted to the built-in REST engine. In other words, the request with a *'datasnap/rest'* path are considered as REST requests. As these are two strings, you can change them to account for different URLs, a topic I'll focus on later on. For now, let's go with standard settings.

The other two components of the web data module provide the DataSnap overall foundation and indicate with class is going to respond to requests (or classes in case you add multiple DSServerClass components). These are the default settings:

```
object DSServer1: TDSServer
    AutoStart = True
    HideDSAdmin = False
end
object DSServerClass1: TDSServerClass
    OnGetClass = DSServerClass1GetClass
    Server = DSServer1
    LifeCycle = 'Session'
end
```

While the DSServer component only needs to get started, manually or automatically, the DSServerClass configuration takes place mostly in the event handler that returns the target class. This has the following default code (generated by the Wizard), which returns the secondary data module class (TServerMethods1), hosted by the corresponding unit (FsrServerClass):

```
procedure TWebModule2.DSServerClass1GetClass(
```

```

    DSServerClass: TDSServerClass;
    var PersistentClass: TPersistentClass);
begin
    PersistentClass := FsrS_ServerClass.TServerMethods1;
end;

```

Finally, the web module provides a default HTTP response for any other action, simply returning some bare-bone HTML:

```

procedure TWebModule2.WebModule2DefaultHandlerAction(Sender: TObject;
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
    Response.Content := '<html><heading/><body>' +
        'DataSnap Server</body></html>';
end;

```

This action is configured at design-time as:

```

Actions = <
    item
        Default = True
        Name = 'DefaultHandler'
        PathInfo = '/'
        OnAction = webModule2DefaultHandlerAction
    end>

```

THE SAMPLE SERVER CLASS GENERATED BY THE DATASNAP WEBBROKER WIZARD

The third unit that is generated by the DataSnap WebBroker Wizard is the sample server class, the class surfacing methods to be called remotely via REST. This is the Delphi class that is connected to the DSServerClass component through the DSServerClass1GetClass event handler listed earlier, and is where most of the actual code will end up.

The skeleton class that gets generated is very simple, and depends on the fact I asked for sample methods in the wizard. Here is the code:

```

type
    TServerMethods1 = class(TDSServerModule)
    private
        { Private declarations }
    public
        function EchoString(Value: string): string;
    end;

```

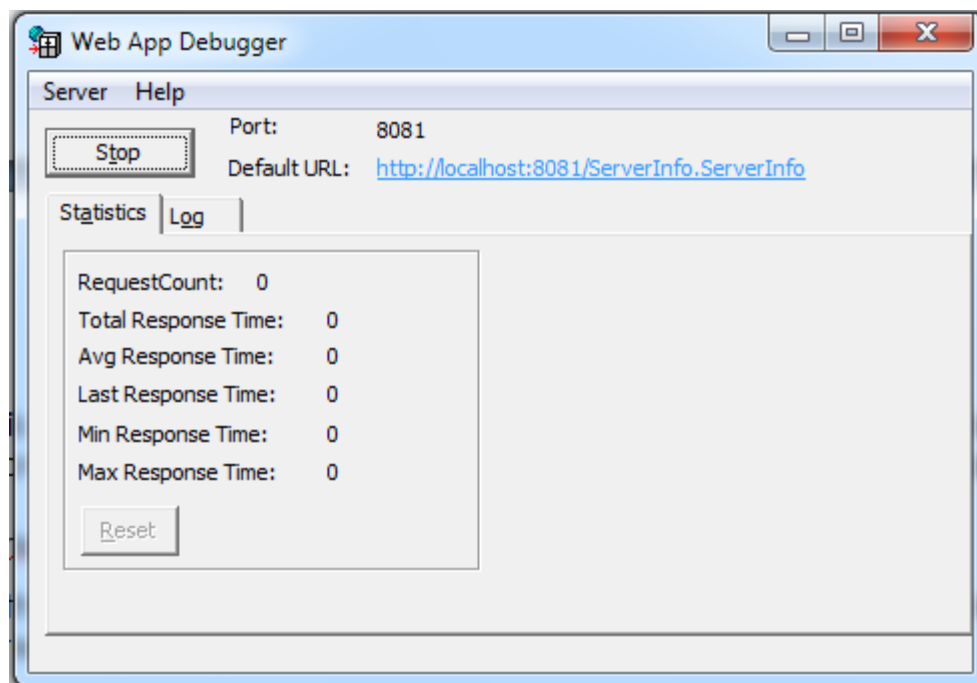
Notice that this class inherits from the `TDSServerModule` class, which is almost a standard data module (with support for `DataSetProvide` components), but compiled with a special compiler option which enables a form of RTTI generation for public methods (predating Delphi 2010 new extended RTTI): `{ $MethodInfo ON }`.

The `EchoString` method by default simply returns the parameter you are passing, but I've updated it slightly to repeat the tail of the string, as in a real "echo":

```
function TServerMethods1.EchoString(Value: string): string;  
begin  
    Result := Value + '...' +  
        Copy (Value, 2, maxint) + '...' +  
        Copy (Value, Length (Value) - 1, 2);  
end;
```

COMPILING AND TESTING THE REST SERVER

We can now compile the server and see if it works. After compiling and running the program, you'll have to run the Web App Debugger (available in Delphi's Tools menu), and start it (using the corresponding button):



The Web App Debugger runs on a specific port, in my configuration 8081 as you can see above. You can open the default URL to see the various applications available in this architecture, or type in the specific URL of our program, which takes the format *ApplicationName.ServerClass*.

In our case the two tokens are identical, so the URL for the server becomes:

`http://localhost:8081/FirstSimpleRestServer.FirstSimpleRestServer`

If you open it in a Web browser you can check if the Web App Debugger and the specific server are running (notice that the specific server must be already running, as the Web App Debugger will not start it automatically). You should see something like:

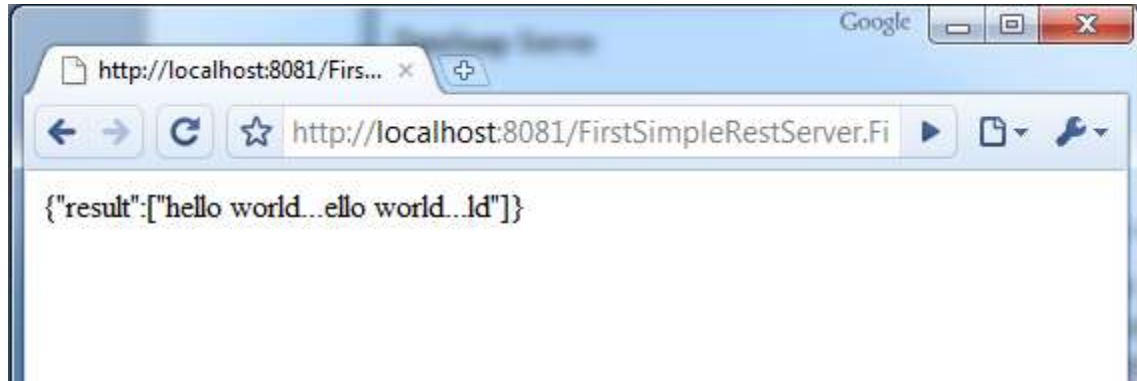


If you remember, this is the HTML returned by the program for the standard action. Not terribly interesting, of course.

The next step is to use the specific URL for the only request our REST server can perform, calling the `EchoString` method of the `TServerMethods1` of the 'rest' support of our 'datasnap' server. The URL is automatically combined by adding the REST server prefix (`/datasnap/rest` by default), the class name, the method name, and the method parameters:

`/datasnap/rest/TServerMethods1/EchoString/hello%20world`

In the URL the `%20` is just a replacement for a space, but you can actually type a space in your browser. Now you'll see the JSON response of our REST server:



Notice that while doing this tests we can use the Web App Debugger for figuring out the actual HTTP requests and responses being transferred. The page above is originated by a browser request:

```
GET /FirstSimpleRestServer.FirstSimpleRestServer/  
  datasnap/rest/TServerMethods1/EchoString/hello%20world HTTP/1.1  
Host: localhost:8081  
Connection: keep-alive  
User-Agent: Mozilla/5.0 (windows; U; windows NT 6.1; en-US)  
  AppleWebKit/532.0 (KHTML, like Gecko) Chrome/3.0.195.27 Safari/532.0  
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,  
  text/plain;q=0.8,image/png,*/*;q=0.5  
Accept-Encoding: gzip,deflate,sdch  
Cookie: LastProgID=FirstSimpleRestServer.FirstSimpleRestServer  
Accept-Language: en-US,en;q=0.8  
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

This request produces the following complete HTTP response:

```
HTTP/1.1 200 200 OK  
Connection: close  
Content-Type: TEXT/HTML  
Content-Length: 44
```

```
{"result":["hello world...ello world...ld"]}
```

As I mentioned earlier, the easy access to this low-level information can be a big bonus when debugging HTTP applications.

CALLING THE REST SERVER FROM A PLAIN DELPHI CLIENT

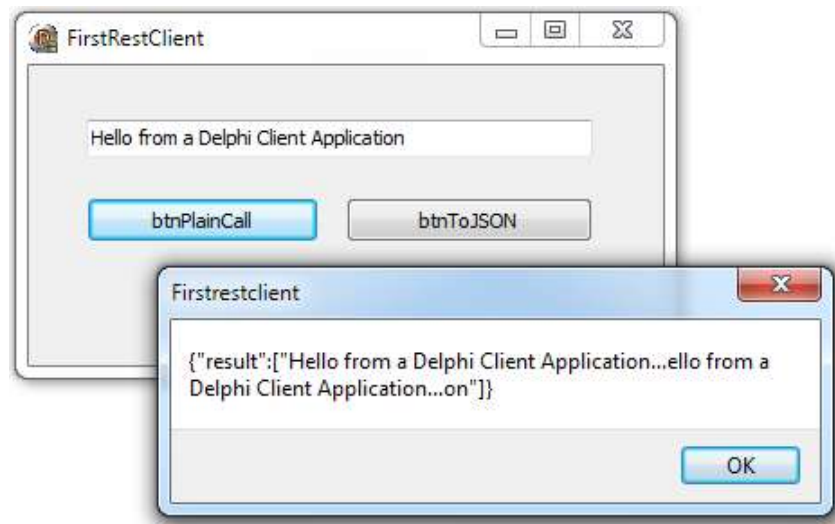
Now that we have built the server and made sure that it works, we can write a Delphi client application to test it. We can use two different approaches. One is to fall back to write a Delphi DataSnap client, using the specific transport layer provided by REST. But it won't make a lot of difference compared to using the HTTP or TCP transport layers of DataSnap.

The second option, which is the one I'm going to follow, is to create a custom REST client just like all of the various clients I built in the first part of this white paper. This means, you could use any other language for building the client application, as we are not relying on any specific support. To accomplish this simply create a standard Delphi VCL application, add an IdHTTP component to it to perform the actual REST request, an edit box for the input, and a button with the code:

```
const
    strServerUrl = 'http://localhost:8081/' +
        'FirstSimpleRestServer.FirstSimpleRestServer/';
    strMethodUrl = 'datasnap/rest/TServerMethods1/EchoString/';

procedure TFormFirstRestClient.btnPlainCallClick(Sender: TObject);
var
    strParam: string;
begin
    strParam := edInput.Text;
    ShowMessage (IdHTTP1.Get(strServerUrl + strMethodUrl + strParam));
end;
```

This call builds a proper URL by concatenating the server address, the relative path to reach the given method with the REST server, and the only parameter. The call results in the following output:



Now what is more interesting is to extract the actual information from the JSON data structure returned by the server. We could use a manual approach, as demonstrated before, but in this case I'd rather take advantage of the JSON support that is available in Delphi 2010.

Delphi new JSON support is made available through a series of classes defined in the DBXJSON, which despite the name can be used even in applications that don't relate to the dbExpress framework. The DBXJSON unit defines classes that you can use to work with the various JSON data types (individual values of different types, arrays, objects, and so on). This is quite useful for customizing the result of server side applications (as I'll show you in the next project) and for reading data received by a client, as in this specific case.

The JSON data that our server returns is a string, but the REST server support creates an object with a named value (or "pair"), and places the actual value in an array. That's why after parsing the result of the HTTP into a JSON data structure, we need to navigate from the object to the pair it contains and from pair to the single element array it holds:

```
procedure TFormFirstRestClient.btnToJSONClick(Sender: TObject);
var
    strParam, strHttpResult, strResult: string;
    jValue: TJSONValue;
    jObj: TJSONObject;
    jPair: TJSONPair;
    jArray: TJSONArray;
begin
    strParam := edInput.Text;
    strHttpResult := IdHTTP1.Get(strServerUrl +
        strMethodUrl + strParam);
    jValue := TJSONObject.ParseJSONValue(
        TEncoding.ASCII.GetBytes(strHttpResult), 0);
    if not Assigned(jValue) then
    begin
        ShowMessage ('Error in parsing ' + strHttpResult);
        Exit;
    end;

    try
        jObj := jValue as TJSONObject;
        jPair := jObj.Get(0); // get the first and only JSON pair
        jArray := jPair.JsonValue as TJSONArray; // pair value is an array
        strResult := jArray.Get(0).value; // first-only element of array
        ShowMessage ('The response is: ' + strResult);
    finally
        jObj.Free;
    end;
end;
```

Again, the complexity is due to the data structure returned by the server, as in other circumstances it would be much easier to parse the resulting JSON and access to it.

CALLING THE REST SERVER FROM AN AJAX WEB APPLICATION

If all you need is to pass objects from a server side Delphi application to another one, there could be many alternatives to using JSON. This choice makes a lot of sense when you want to call the Delphi compiled server from a JavaScript application running in the browser. This case is quite relevant because AJAX (Asynchronous JavaScript calls done in the Web browser) was and still is one of the driving forces behind the adoption of REST. Calling a corresponding SOAP server from a Browser based program is incredibly more complicated.

So, how can we create an application mimicking the client I just wrote but running in the Web browser. I could have used many different approaches and libraries, but my preference at this time goes to using jQuery, an incredibly powerful open source JavaScript library available at:

<http://jquery.com>

As don't have time to delve into jQuery and its usage in this paper, but I'll at least try to explain the jQuery code behind this specific example. First of all, the HTML page includes jQuery and its JSON support:

```
<head>
  <title>jQuery and Delphi 2010 REST</title>
  <script
    src="http://jqueryjs.googlecode.com/files/jquery-1.3.2.min.js"
    type="text/javascript"></script>
  <script
    src="http://jquery-json.googlecode.com/files/jquery.json-2.2.min.js"
    type="text/javascript"></script>
</head>
```

Second, the page has a very simple user interface, with some text, an input field and a button (without any sophisticated CSS and added graphics, as I really wanted to keep this focused):

```
<body>
  <h1>jQuery and Delphi 2010 REST</h1>

  <p>This example demonstrates basic use of jQuery calling a
    barebone Delphi 2010 REST server. </p>

  <p>Insert the text to "Echo":

  <br/>
  <input type="text" id="inputText" size="50"
    value="This is a message from jQuery">
```

```

<br/>
<input type="button" value="Echo" id="buttonEcho">

<div id="result">Result goes here: </div>
</body>

```

If this is the skeleton, let us now look at the actual JavaScript code. What we have to do is add an event handler to the button, read the input text, make the REST call, and finally display the result. To access to the page objects, I'm going to use the simplest of jQuery selectors, based on the objects ID, as in:

```
$("#inputText")
```

This returns a jQuery object wrapping the input text DOM element. To define an event handler we can pass an anonymous method parameter to the click() function of the button. Two more calls are the REST call itself (using the global getJSON) and the html() call to add the result to the HTML of the output element.

This is the complete code at the heart of this demo, a very compact but not exactly readable JavaScript snippet:

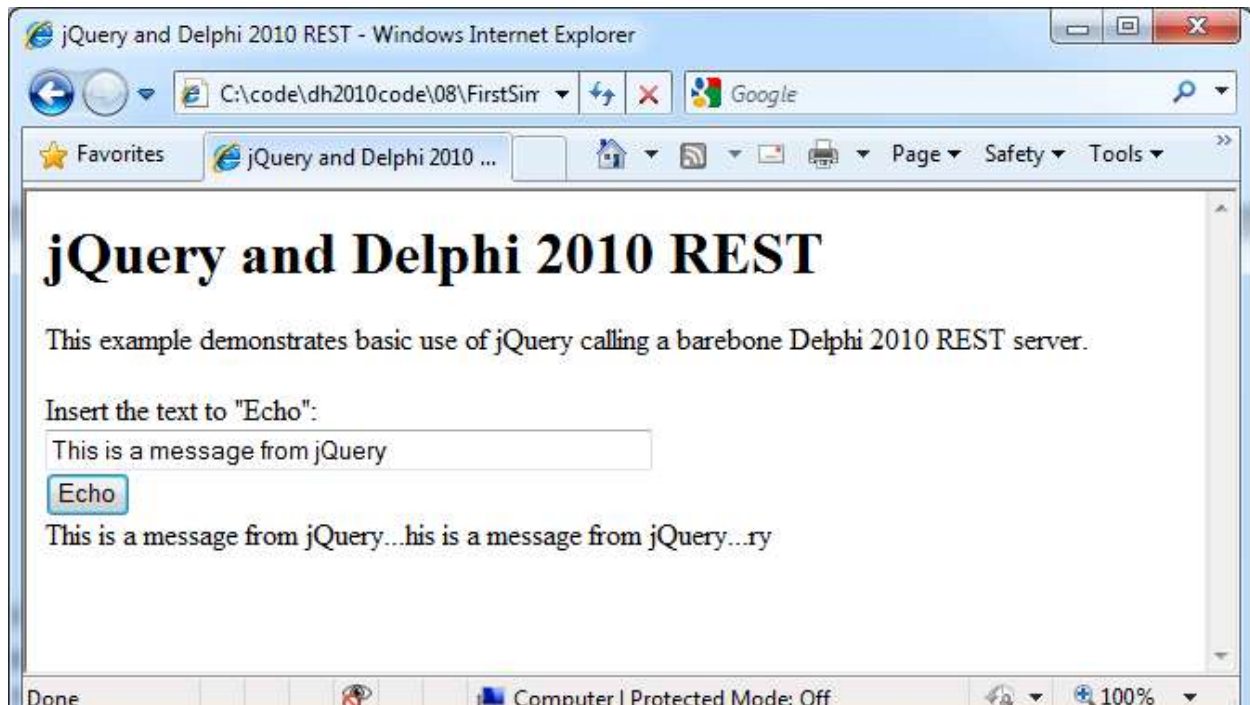
```

$(document).ready(function() {
    $("#buttonEcho").click(function(e) {
        $.getJSON("http://localhost:8081/"
            "FirstSimpleRestServer.FirstSimpleRestServer/"
            "datasnap/rest/TServerMethods1/EchoString/" +
            $("#inputText").val(),
            function(data) {
                $("#result").html(data.result.join(''));
            } );
    });
});

```

Just by opening an HTML file with the given code you can call the custom server, but only if the browser permission settings allow an AJAX call from a local file to a local REST server. In general, most browsers will only let you call REST servers on the same site originating the HTML page.

In any case, Internet Explorer seems to work fine on this local file, after enabling local scripts and asking for limited security (available since the file is on the local machine, see the icons in the status bar):



On other browsers, you need to have the web server returning both the HTML page and the REST data, which is not a terribly big deal as our REST server is indeed a web server. So, all I had to do for a "server side" solution (as far as the web browser is concerned) is to add an action to the web server module (which I hooked to the "/file" URL) and return the HTML file from it:

```
procedure TwebModule2.WebModule2WebActionItem1Action(Sender: TObject;  
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);  
var  
    strRead: TStreamReader;  
begin  
    strRead := TStreamReader.Create('jRestClient.html');  
    try  
        Response.Content := strRead.ReadToEnd;  
    finally  
        strRead.Free;  
    end;  
end;
```

Now we can refer to given server page with the /file URL, get the file with the JavaScript code, and let it call our REST server:



The difference between this and the previous image is not just that I'm using a different browser, but that I'm pointing to a different URL. Rather than loading a file, in this second case I'm using the server side REST application as a full Web server, returning the HTML used for calling the same server via AJAX.

RETURNING OBJECTS AND UPDATING THEM

Now that we have explored the development of a very simple REST server with Delphi 2010 DataSnap support, it is time to try to figure out the actual code we can write on the server to make it more powerful. As we have seen, the server returns JSON data, converting the result of your functions to this format. We can pass an object as result and have it converted. However, in most practical situations, it would be better to take full control and create specific JSON objects on the server side and return them. That is going to be one of the goals of our next project.

The same project will also show how to process other HTTP methods beside *get*, letting us retrieve but also modify a server side object from a simple Browser-based client written in JavaScript. Finally, in doing this we'll focus on URL management and figure out how to make them nicer and more flexible.

RETURNING JSON OBJECTS AND VALUES

For this second project I've used the DataSnap WebBroker Wizard, picked the Web App Debugger architecture (again), and decided to use a TPersistent base class, as I don't need a data module as target class for the REST calls. As you need specific RTTI support, the rule is to inherit (at least) from TPersistent and mark the class with the *\$METHODINFO* directive, as in the following generated code:

```
{ $METHODINFO ON }
```

```

type
  TObjectsRest = class(TPersistent)
  public
    function PlainData (name: string): TJSONValue;
    function DataMarshal (name: string): TJSONObject;
  end;
{$METHODINFO OFF}

```

As you can see I've added a couple of functions to the class for returning either a value or a full object. Later I'll add other methods to the class.

The data structure behind this application is a list of objects of a custom type (which could have been written in a more object-oriented way, but I wanted to keep simple for the sake of the example):

```

type
  TMyData = class (TPersistent)
  public
    Name: String;
    Value: Integer;
  public
    constructor Create (const aName: string);
  end;

```

The objects are kept in a dictionary, implemented using the generic container class TObjectDictionary<TKey,TValue> defined in the Generics.Collections unit since Delphi 2009. This global object is initialized when the program starts with the addition of a couple of predefined objects. Notice that I use a specific AddToDictionary procedure to add the objects, to make sure the object name is in sync with the dictionary key and it has a random value if none is provided:

```

var
  DataDict: TObjectDictionary <string,TMyData>;

procedure AddToDictionary (const aName: string; nVal: Integer = -1);
var
  md: TMyData;
begin
  md := TMyData.Create (aName);
  if nVal <> -1 then
    md.Value := nVal;
  DataDict.Add(aName, md);
end;

initialization
  DataDict := TObjectDictionary <string,TMyData>.Create;

```



```
AddToDictionary( 'Sample ');
```

Having this data structure in place, we can now focus on the first two sample methods used to return the JSON values. The first returns the value of the given object (picking a default one if no parameter is passed to the function):

```
function TObjectsRest.PlainData(name: string): TJSONValue;  
begin  
    if Name = '' then  
        name := 'Sample'; // default  
    Result := TJSONNumber.Create(DataDict[name].Value);  
end;
```

If we use an URL with or without the parameter (as in the following two lines):

```
/datasnap/rest/TObjectsRest/PlainData/Test  
/datasnap/rest/TObjectsRest/PlainData
```

we will still get a resulting JSON response, either for the specific object or a default one:

```
{"result": [8978]}
```

What if we want to return a complete object rather than a specific value? Our REST server cannot return a TObject value, as the system has no way to convert it automatically, but it can indeed use the new JSON marshaling support for converting an existing object to the JSON format:

```
function TObjectsRest.DataMarshal(name: string): TJSONObject;  
var  
    jMarshal: TJSONMarshal;  
begin  
    jMarshal := TJSONMarshal.Create(TJSONConverter.Create);  
    Result := jMarshal.Marshal(DataDict[name]) as TJSONObject;  
end;
```

This approach is mostly useful when you need to recreate the object in the Delphi client application, while it is not particularly handy in the case where the client is written in another language. The resulting JSON looks a little ugly:

```
{"result": [{  
    "type": "ObjectsRestServer_Classes.TMyData",  
    "id": 1,  
    "fields": {  
        "Name": "Test",  
        "Value": 8068}  
}]}
```

```
}}}
```

So, what would be the best option to return a JSON object? I think it would be to create one on the server side, using the support classes. This is what I've done in the MyData function:

```
function TObjectsRest.MyData(name: string): TJSONObject;  
var  
    md: TMyData;  
begin  
    md := DataDict[name];  
    Result := TJSONObject.Create;  
    Result.AddPair(  
        TJSONPair.Create ('Name', md.Name));  
    Result.AddPair(  
        TJSONPair.Create ('Value',  
            TJSONNumber.Create(md.Value)));  
end;
```

As you can see I've created a TJSONObject and added *two pairs or properties* for the name and the value. I could have used a dynamic name (that is, used the name for the name part of the pair), but this would have made it harder to retrieve the data on the client side. The result of this code should look like the following cleaner JSON code:

```
{"result": [{  
    "Name": "Test",  
    "Value": 8068  
}]}
```

LISTING OBJECTS WITH TJSONARRAY

Now having a list of objects, you might well need to access the list of objects. Having a list with the names only (and no data) will be useful when building a client side user interface.

For returning a list you can use a TJSONArray, which in this case will be an array of strings, which I create using an enumerator on the Keys of the dictionary:

```
function TObjectsRest.List: TJSONArray;  
var  
    str: string;  
begin  
    Result := TJSONArray.Create;  
    for str in DataDict.Keys do  
        begin  
            Result.Add(str);  
        end;  
    end;
```

end;

The result of this call is an array in JSON format, which in turned is passed (as usual) in an array called result (hence the double nested array notation):

```
{"result": [
  ["Test", "Sample"]
]}
```

Now that we have a way to return a list of values and fetch the data of each individual element, we can start building a user interface.

STARTING WITH THE CLIENT: LIST AND VALUES

Rather than having to build the initial HTML with the list of values, to let the user pick one, we can fully exploit the AJAX model.

The page on start up will have no data at all, only the HTML elements and the JavaScript code. As soon as the page is loaded, even without user intervention, it will reach the server asking for the actual data and populating the user interface.

As an example, on start up the program shows the value of the Sample object, using the following HTML elements and AJAX call (executed as when document is ready, that is the DOM has finished loading):

```
<div>Sample: <span id="sample"></span></div>

<script>
  var baseUrl = "/ObjectsRestServer.RestObjects/dataSnap" +
    "/rest/TObjectsRest/";

$(document).ready(function() {
  $.getJSON(baseUrl + "MyData/Sample",
    function(data) {
      strResult = data.result[0].Value;
      $("#sample").html(strResult);
    } );
});
```

The AJAX call to MyData passes the object name as a further URL parameter and extracts from the result array the property/pair called Value, showing it in an empty span HTML element. Something similar (but somewhat more complex) happens for the list. Again, there is an AJAX call, but this time we have to build the resulting HTML. The operation is performed in a separate *refreshList* function called both automatically at startup and manually by the user:

```
<div>Current entries list:
  <a href="#" id="refresh">Refresh</a>
  <span id="list"></span></div>
```

```

function refreshList()
{
    $.getJSON(baseUrl + "list",
        function(data) {
            thearray = data.result[0];
            var ratingMarkup = ["<br>"];
            for (var i=0; i < thearray.length; i++) {
                ratingMarkup = ratingMarkup +
                    "<a href='#>" + thearray[i] + "</a><br>";
            }
            $("#list").html(ratingMarkup);
        } );
};

```

The code uses a for loop to scan the resulting array. I could have used the *\$.each* enumeration mechanism of jQuery, but this would have made the code more complex to read. The for loop creates the HTML, which is later displayed in the span placeholder with the given ID. This is a sample output with the value of the Sample object (the code shown earlier) plus the list of the values returned in the JSON array:



As I mentioned earlier, the *refreshList* function is called at startup (in the ready event handler) and also connected with a corresponding link, so that the user can later refresh the data of the list without having to refresh the entire HTML page:

```

$(document).ready(function() {

    refreshList();

```

```

$("#refresh").click(function(e) {
    refreshList();
} );

```

There is actually a little more to the code generation. As soon as we have the HTML for the list, which is a list of links, we need to hook code to those links so that selecting each entry of the list the client application will load the corresponding server side object. The user interface for the object data is made of two input boxes, which will later use also for manipulating the object data. The behavior is added to each anchor within the *list* container.

```

$("#list").find("a").click(function(e) {
    var wasclicked = $(this);
    $.getJSON(baseUrl + "MyData/" + $(this).html(),
        function(data) {
            strResult = data.result[0].value;
            $("#inputName").val(wasclicked.html());
            $("#inputValue").val(strResult);
        } );
});

```

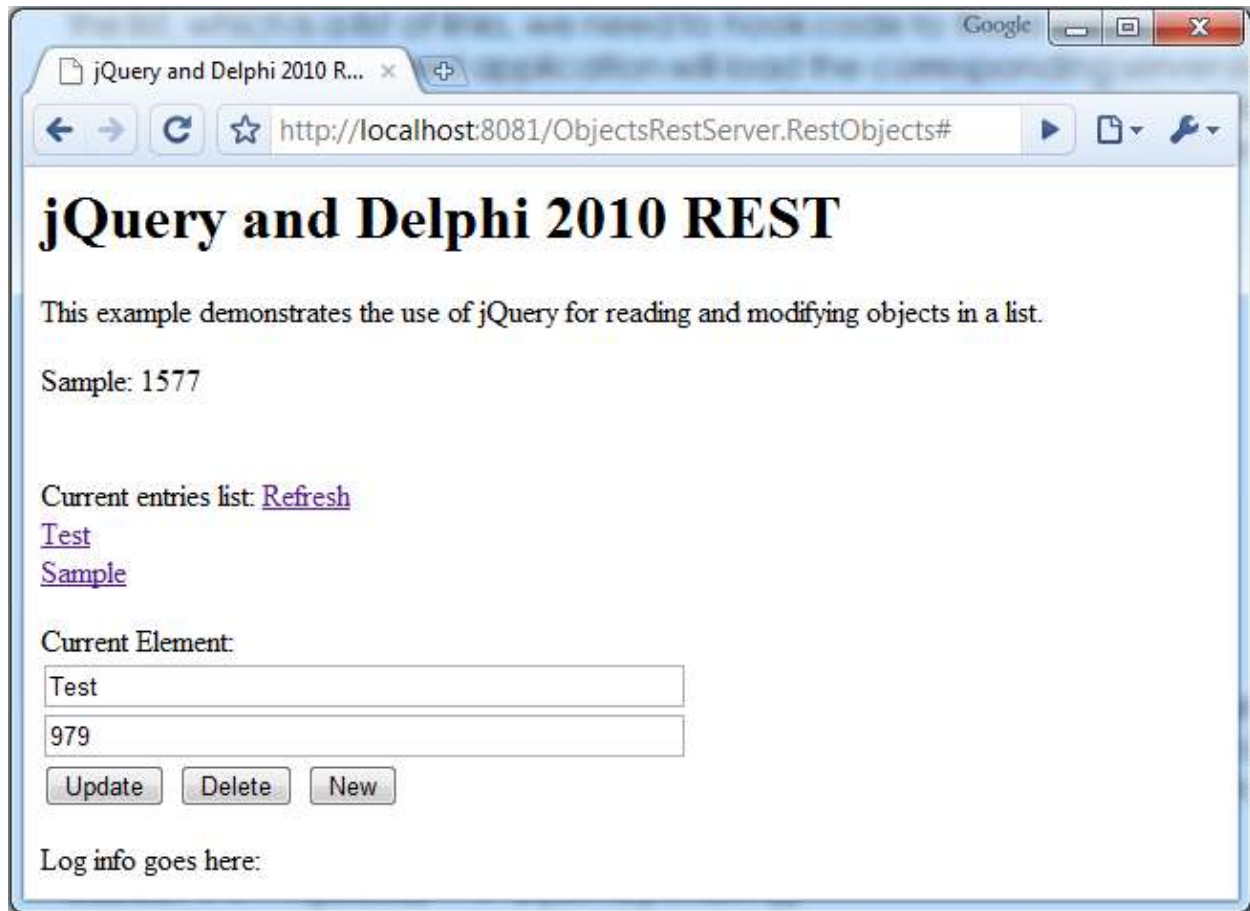
Notice the use of the `$(this)` call, which is more or less the Sender parameter for a Delphi event. The html content of the element that was clicked is its text, which is the name of the element we have to pass to the server in the URL, with the expression:

```

baseUrl + "MyData/" + $(this).html()

```

Now with this code in place we can see the effect of clicking on one of the elements of the list: A further AJAX call will reach our server asking for a given value, and the returned value is displayed in two input text boxes:



As you can see above, the program let us retrieve a value, but has also three buttons to perform the most common operations (the so called CRUD interface – Create, Read, Update, Delete). This is supported in HTML using the 4 code HTML methods, respectively PUT, GET, POST, and DELETE. How these are supported by a Delphi 2010 REST server is the subject of the next section.

POST, PUT, AND DELETE

Up to now we have seen only how to get data from our REST server, but what about updating it? The generally agreed idea in REST is to avoid using specific URLs for identifying the operations, but use an URL only to identify server side objects (like MyData/Sample in our case) and use the HTTP methods to indicate what do to.

Now if Delphi's REST support simply mapped URLs to methods, we would have been out of luck. Instead, it maps URLs plus the HTTP method to methods, using a rather simple scheme: the name of the operation is prepended to the method name, using the following mapping:

- GET get (default, can be omitted)

- POST update
- PUT accept
- DELETE cancel

You can customize these mappings by handling the four corresponding event handlers of the DSHTTPWebDispatcher component. If we decide to go with the standard naming rules, to support the various operations we need to define our server class as:

```
type
TObjectsRest = class(TPersistent)
public
    function List: TJSONArray;
    function MyData (name: string): TJSONObject;
    procedure updateMyData (name, value: string);
    procedure cancelMyData (name: string);
    procedure acceptMyData (name, value: string);
end;
```

To get a delete an element we only need the name, while to create or update it we need a second parameter with the data. The implementation of the three new methods is rather simple and direct, also because they don't need to return a value (needless to say I should have checked that parameters are not empty and that the server side object really exists):

```
procedure TObjectsRest.updateMyData (name, value: string);
begin
    DataDict[name].Value := StrToIntDef (Value, 0);
end;

procedure TObjectsRest.cancelMyData(name: string);
begin
    DataDict.Remove(name);
end;

procedure TObjectsRest.acceptMyData(name, value: string);
begin
    AddToDictionary (name, StrToIntDef (Value, 0));
end;
```

CLIENT SIDE EDITING

Now that we have the CRUD operations available on the REST server, we can complete our JavaScript client application with the code of the three editing buttons (the image with the browser-based the user interface was displayed earlier).

While jQuery has specific support for the get operation (with different versions, including the JSON-specific one we have used earlier) and some support for post operations, for the other HTTP methods you have to use the lower level (and slightly more complex \$.ajax call. This call has as a parameter a list of paired values, out of over a dozen that are possible. The more relevant parameters are the type and the url, while data lets you pass further POST parameters.

Posting our update is rather simple, and we can provide all of the data to our REST server using the URL:

```
$("#buttonUpdate").click(function(e) {
    $.ajax({
        type: "POST",
        url: baseUrl + "MyData/" +
            $("#inputName").val() + "/" + $("#inputValue").val(),
        success: function(msg){
            $("#log").html(msg);
        }
    });
});
```

Deleting is equally simple, as we need to create the URL with the reference to the object we want to remove:

```
$("#buttonDelete").click(function(e) {
    $.ajax({
        type: "DELETE",
        url: baseUrl + "MyData/" + $("#inputName").val(),
        success: function(msg){
            $("#log").html(msg);
        }
    });
});
```

It took me some more time to figure out how to implement PUT, as if you don't provide any data, some browsers (notably Chrome) will post the data as "undefined" and this will make the HTTP input parsing of the REST server crash with an error. As we need to pass information (and we cannot pass more parameters than the server requires, which will be equally flagged as an error), what we can do is replace one of the URL elements with a corresponding data element:

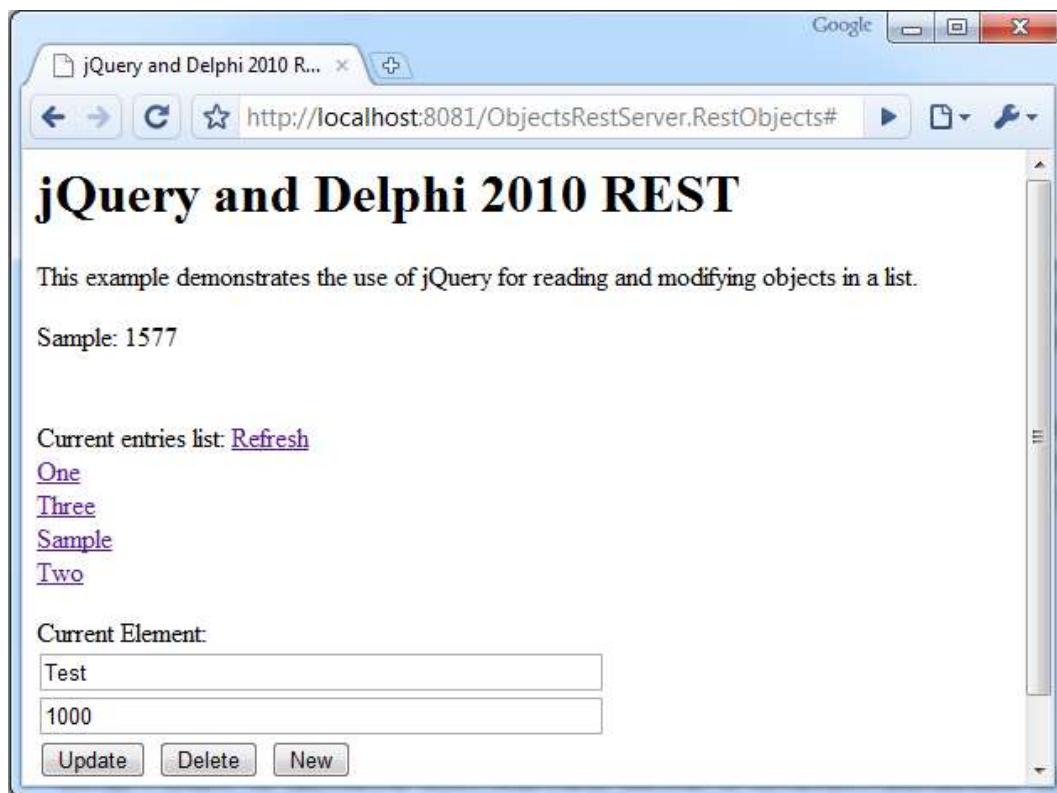
```
$("#buttonNew").click(function(e) {
    $.ajax({
        type: 'PUT',
        data: $("#inputValue").val(),
        url: baseUrl + "MyData/" + $("#inputName").val(),
        success: function(msg){
            $("#log").html(msg);
        }
    });
});
```



```
    }  
    });  
});
```

Notice that jQuery documentation specifically warns against using PUT in browsers, as you might get mixed results. That might as well be the reason for which a number of REST services (including those from Microsoft) tend to use POST for both updating and creating server side objects. I prefer keeping the two concepts separate, for clarity and consistency, whenever possible.

So with the three extra methods added to our class and the proper JavaScript calls, we now have an example with a complete Browser-based user interface for creating and editing objects in our REST server. Here is an example with a few objects created:



DATA-ORIENTED REST SERVERS

If the original idea behind DataSnap is focused on moving data tables from a middle-tier server to a client application, it might be quite odd at first to realize that you cannot return a dataset from a REST server written in Delphi 2010. Well, you cannot return it directly or as easily as you return its XML representation, but you can create a JSON result with all of the data of a Delphi dataset. That's the focus my last example.

The program is quite bare bone, as all it does is return the data of a complete Dataset, with no metadata. It could be extended in several ways and lacks a polished user interface, but should get you started. The server class has only one method, returning an entire dataset in a JSON array, with individual objects / records inside it:

```
function TServerData.Data: TJSONArray;
var
    jRecord: TJSONObject;
    I: Integer;
begin
    ClientDataSet1.Open;
    Result := TJSONArray.Create;

    while not ClientDataSet1.EOF do
    begin
        jRecord := TJSONObject.Create;
        for I := 0 to ClientDataSet1.FieldCount - 1 do
            jRecord.AddPair(
                ClientDataSet1.Fields[I].FieldName,
                TJSONString.Create (ClientDataSet1.Fields[I].AsString));
        Result.AddElement(jRecord);
        ClientDataSet1.Next;
    end;
end;
```

This method is invoked by the client application after loading the page, building an HTML table dynamically, with the following jQuery code (you should have become familiar with the coding style by now):

```
$(document).ready(function() {

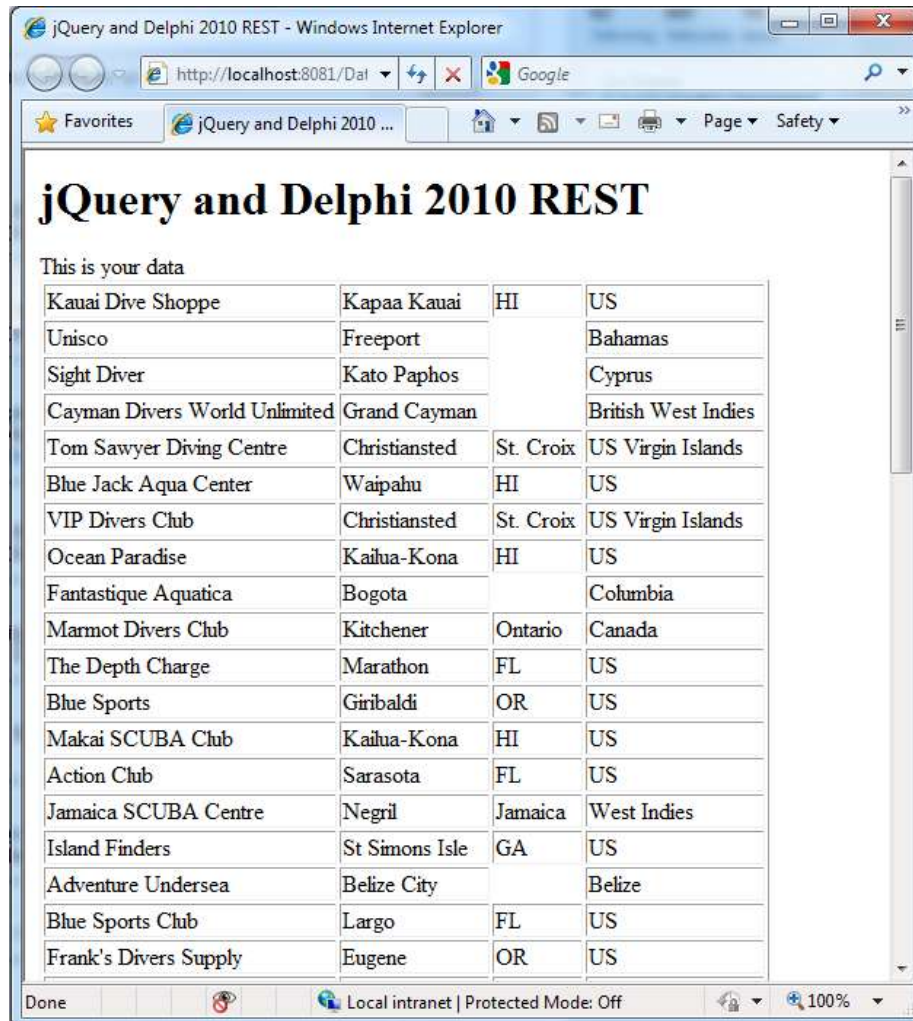
    $.getJSON(
        "/DataRestServer.RestDataServer/datasnap/rest/TServerData/Data",
        function(data) {
            thearray = data.result[0];
            var ratingMarkup = "<table border='1'>";
            for (var i=0; i < thearray.length; i++) {
                ratingMarkup = ratingMarkup + " <tr><td>" +
                    thearray[i].Company +
                    "</td><td>" +
                    thearray[i].City +
                    "</td><td>" +
                    thearray[i].State +
                    "</td><td>" +
                    thearray[i].Country +
                    "</td></tr>";
            }
        }
    );
});
```

```

    }
    ratingMarkup = ratingMarkup + "</table>";
    $("#result").html(ratingMarkup);
  } );
});

```

The bare-bone result is visible below in Internet Explorer:



Can we improve it a little bit? What about avoiding listing all of the fields we want to display in the JavaScript, in case we want them all? The final version of the program adds some metadata support to improve the final output.

On the server side, there is a second function returning an array of field names from the dataset field definitions:

```

function TServerData.Meta: TJSONArray;
var
  jRecord: TJSONObject;
  I: Integer;
begin
  ClientDataSet1.Open;
  Result := TJSONArray.Create;

  for I := 0 to ClientDataSet1.FieldDefs.Count - 1 do
    Result.Add(ClientDataSet1.FieldDefs[I].Name);
end;

```

The client-side JavaScript has been expanded with a second call to the REST server to get the metadata:

```

$.getJSON(
  "/DataRestServer.RestDataServer/datasnap/rest/TServerData/Meta",
  function(data) {
    theMetaArray = data.result[0];
  }
);

```

This information is used to create the table header and to access to the object properties dynamically, with the notation *object["propertyname"]*. In our case the existing code used to access to an object with the property symbol:

```
thearray[i].Company
```

becomes the following code that reads the property by name, using the name of the field stored in the metadata:

```
thearray[i][theMetaArray[j]].
```

This is the complete JavaScript code used to create the HTML markup:

```

var ratingMarkup = "<table border='1'><tr>";
// header
for (var j=0; j < theMetaArray.length; j++) {
  ratingMarkup = ratingMarkup + "<th>" +
    theMetaArray[j] + "</th>";
};
ratingMarkup = ratingMarkup + "</tr>";

// content
for (var i=0; i < thearray.length; i++) {
  ratingMarkup = ratingMarkup + "<tr>";
  for (var j=0; j < theMetaArray.length; j++) {

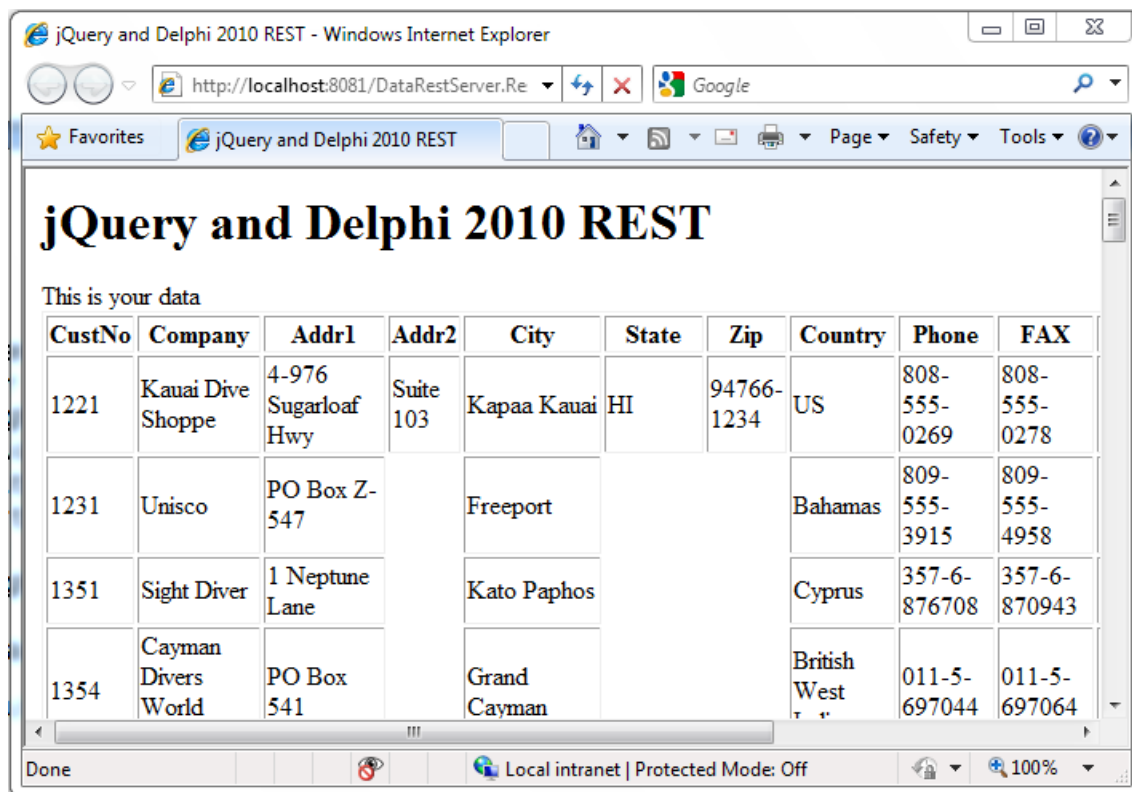
```

```

        ratingMarkup = ratingMarkup + "<td>" +
            thearray[i][theMetaArray[j]] + "</td>";
    };
    ratingMarkup = ratingMarkup + "</tr>";
}
ratingMarkup = ratingMarkup + "</table>";

```

The output of this extended version becomes slightly nicer (and more flexible):



Again, this serves only as a starting point, and I didn't use any of the jQuery plug-ins, which would add significant capabilities to HTML tables, turning them into powerful user interface grids, with sorting, filtering, and editing capabilities.

CONCLUSION

In this white paper, I have just scratched the surface of the REST architecture and of the development of REST client and server applications with Delphi 2010. I've just provided a very limited overview of related technologies (XML, JSON, XPath, JavaScript, jQuery) which you might need to learn in some detail to become proficient on REST architectures.

With the growing number of public REST servers, with the emerging of cloud-computing, with the growing interest for web hosted applications, Delphi can play a significant role both in the

development of rich user interface clients calling into remote servers and in the development of actual servers for manipulating data structures in a client application (written in any language) or directly in the browser.

As the final demos showed, combining JavaScript and a Delphi REST server you'll be able to use the Embarcadero IDE for the development of professional, high-quality, and modern web based applications.

ABOUT THE AUTHOR

Marco is the author of the best-selling Mastering Delphi series and in the recent years he has self-published books on the latest versions of Delphi, including his Delphi 2007 Handbook, Delphi 2009 Handbook, and Delphi 2010 Handbook (which is being completed at this time).

Beside training and consulting on Delphi, Marco is available for consulting on Web architectures and the integration of Delphi projects with the web, but in terms of calling other servers and exposing them to the world.

You can read Marco's blog at <http://blog.marcocantu.com>, follow him on Twitter at <http://twitter.com/marcocantu>, and contact him on marco.cantu@gmail.com.

Marco wishes to thank Daniele Teti for his help in working on this white paper, specifically on Delphi 2010's JSON support.



Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance and accelerate innovation. The company's flagship tools include: Embarcadero® Change Manager™, Embarcadero™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.