

## REST Servers in Delphi XE Using DataSnap

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

December 2010

---

**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

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. Delphi 2010 added REST support through the DataSnap infrastructure and Delphi XE pushes the model even further with support for WebBroker integration and the creation of JavaScript proxies for the methods exposed by the web service.

This paper delves into the development of REST servers in Delphi XE, showing how you can use the features available in the box, how to extend them with extra Delphi support code, and how to take advantage of the jQuery library. Within the various models offered by DataSnap, our specific focus will be the development of Delphi REST servers, coupled with browser-based applications written in JavaScript and based on jQuery. Keep in mind, however, that the same REST servers could also be used by a variety of clients, including mobile device-based applications.

## INTRODUCTION: THE CONCEPTS BEHIND REPRESENTATIONAL STATE TRANSFER

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, and between web sites and business databases – 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. At the same time, this server-side processing can be leveraged by custom client applications and also by applications running natively in Web browsers.

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 Simple Object Access Protocol (SOAP) referenced at the site at <http://www.w3.org/TR/soap/>. Incidentally, Delphi has had support for SOAP for several years now, with recent

extensions in Delphi XE. Another web service solution is the use of a REST (Representational State Transfer) approach.

In this paper, we'll focus exclusively on REST, because it is more flexible and best suited as a back-end for client browser applications (but also in many other scenarios). 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 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.

In contrast to SOAP, REST architectures use HTTP and its data format (generally XML or JSON) 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 not a REST standard and that you need no specific tools for REST development, there are existing standards that REST relies upon and those are worth introducing shortly (an in-depth description of each could take an entire book). The specific focus here is Delphi support for these technologies.

The HyperText Transfer Protocol is the standard at the heart of the World Wide Web, and needs no introduction. Granted, HTTP can be used not only 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. Because the Indy suite uses blocking threads, the user interface of your program will be stuck until the requests are returned (which may take a long time in the 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. You can create a standalone web server using the `IdHttpServer`

component or you can create web server extensions (CGI applications, ISAPI, or Apache modules). Starting with Delphi XE, the WebBroker architecture supports both models (while in the past it was limited to web server extensions). In fact, as you create a DataSnap REST Server with the new Delphi XE Wizard, you will obtain a standalone web server, indirectly based on the `IdHttpServer` component.

## JSON IN DELPHI

While web services use either XML or JSON, Delphi REST servers default to the latter. That's why I want to end this introduction to Delphi and REST by providing a minimal introduction to the JavaScript Object Notation, or JSON.

### THE JSON NOTATION

JSON is a text-based notation used to represent JavaScript objects, so that they can be made persistent or transferred from one application to another (or from one computer to another). While a few years ago the consensus was to use XML as a notation to represent complex data structures, in the last few years JSON has gained popularity because it is more compact, more tied to programming language concepts, and very easy to parse in JavaScript browser based applications... and by most other programming languages due to a growing set of libraries and tools.

You can learn more about JSON by reading the RFC 4627 specification of the IEFT (Internet Engineering Task Force) or looking to the *official* JSON home page:

<http://www.ietf.org/rfc/rfc4627.txt>  
<http://json.org>

You can also keep reading for a short introduction, as JSON is relatively simple to understand, with four primitive types and two structures. The primitive types are numbers, strings, Booleans (true or false) and the null value.

The two JSON data structures are:

- JSON Objects - Collections of name and value pairs, enclosed in curly braces and separated by commas (while the two elements of each pair are divided by a colon). The collections represent records or objects.
- JSON Arrays - Lists of values within square brackets and separated by commas. The lists represent arrays or collections.

Here are simple examples of the two notations, an object with two *pairs* (all strings including pair names are indicated by double quotes) and a list of two values, a number and a string:

```
{ "Name": "Marco",  
  "Value": 100  
}
```

```
[22, "foo"]
```

Of course, you can combine these constructs at will, so you can use objects and arrays as values of a pair and as elements of an array:

```
{ "Me": {  
    "FirstName": "Marco",  
    "LastName": "Cantù",  
    "Wife": "Lella",  
    "Kids": [  
        { "Name": "Benedetta",  
          "Age": 11  
        },  
        { "Name": "Jacopo",  
          "Age": 7  
        }  
    ]  
  }  
}
```

## JSON IN DELPHI 2010

While there have been a few JSON libraries for Delphi in the past, the first version with native support is Delphi 2010. The native JSON support has been made available through a series of classes defined in the DBXJSON unit, which (despite the name) can be used even in applications that don't relate at all with 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, pairs, and objects) all inheriting from *TJSONValue*:

- Primitive values include *TJSONNull*, *TJSONFalse*, *TJSONTrue*, *TJSONString* and *TJSONNumber*.
- Data structures include *TJSONObject* (and the internal *TJSONPair*) and *TJSONArray*.

Here is a simple code fragment, extracted from the *JsonTests* project, used to demonstrate the output of the different primitive types. Notice that each temporary object you create must be manually freed, hence the idea of adding the *LogAndFree* private support method:

```
procedure TFormJson. LogAndFree (j Val ue: TJSONVal ue);
begin
  try
    Log (j Val ue. Cl assName + ' > ' + j Val ue. ToStri ng);
  finally
    j val ue. Free;
  end;
end;
```

```
procedure TFormJson. btnVal uesCl i ck(Sender: TObj ect);
begin
  LogAndFree (TJSONNumber. Create(22));
  LogAndFree (TJSONStri ng. Create('sample text'));
  LogAndFree (TJSONTrue. Create);
  LogAndFree (TJSONFal se. Create);
  LogAndFree (TJSONNul l . Create);
end;
```

This is the corresponding output:

```
TJSONNumber > 22
TJSONStri ng > "sample text"
TJSONTrue > true
TJSONFal se > false
TJSONNul l > nul l
```

It is equally simple to use the other classes of the DBXJSON unit for creating arrays and objects. An array is a structure to which you can add any value (including arrays and objects):

```
procedure TFormJson. btnSi mpl eArrayCl i ck(Sender: TObj ect);
var
  j Li st: TJSONArray;
begin
  j Li st := TJSONArray. Create;
  j Li st. Add(22);
  j Li st. Add('foo');
  j Li st. Add(TJSonArray. Create (TJSONTrue. Create));
  (j Li st. Get (2) as TJSonArray). Add (100);
  Log (j Li st. ToStri ng);
  j Li st. Free;
end;
```

The JSON output shows the two nested arrays as follows:

```
[22, "foo", [true, 100]]
```

Note that the JSON containers (arrays and objects) own their internal elements, so you can free the container to clean up the memory for the entire group of JSON values.

When you have an object, the only element you can add to it is a pair, but the value of the pair can be any JSON value, including a nested object:

```
procedure TFormJson. btnSimpleObjectClick(Sender: TObject);
var
  jsonObj, subObject: TJSONObject;
begin
  jsonObj := TJSONObject.Create;
  jsonObj.AddPair(TJSONPair.Create ('Name', 'Marco'));
  jsonObj.AddPair(TJSONPair.Create ('Value',
    TJSONNumber.Create(100)));

  subObject := TJSONObject.Create(
    TJSONPair.Create ('Subvalue', 'one'));
  jsonObj.AddPair(
    TJSONPair.Create ('Object', subObject));

  Log (jsonObj.ToString);
  jsonObj.Free;
end;
```

The JSON representation of this object (with a little manual formatting to improve readability) is the following:

```
{ "Name": "Marco",
  "Value": 100,
  "Object": {
    "Subvalue": "one"
  }
}
```

## PARSING JSON IN DELPHI

Creating JSON data structures using the DBXJSON classes and generating the corresponding JSON representation is interesting, but it is even more interesting to know you can do the reverse, that is parse a string with a JSON representation to create the corresponding Delphi objects.

Once you have a JSON string, you can pass it to the *ParseJSONValue* class method of the *TJSONObject* class, which returns a *TJSONValue* object. In cases where we know the



evaluation returns a JSON object we have to cast it back to the proper type. The *ParseJSONValue* class method doesn't accept a string as parameter, but requires an array of bytes with an ASCII or UTF8 encoding (this last feature was not included in Delphi 2010 and is new in version XE). In both cases, we need to take the string and turn it into an array of bytes using the proper Encoding class:

```
TEncoding.ASCII.GetBytes(strParam) // a string converted to ASCII
TEncoding.UTF8.GetBytes(sUtf8) // any UTF8 (or Unicode) string
```

In case of a UTF8 string, you have to pass an additional parameter to the *ParseJSONValue* method or call the sibling *ParseJSONValueUTF8* method. Overall, the initial portion of the *btnParseObjClick* (still part of the main form of the *JsonTests* example) is the following:

```
var
  strParam: string;
  jsonObj: TJSONObject;
begin
  strParam := '{"value": 3}';
  jsonObj := TJSONObject.ParseJSONValue(
    TEncoding.ASCII.GetBytes(strParam), 0) as TJSONObject;
```

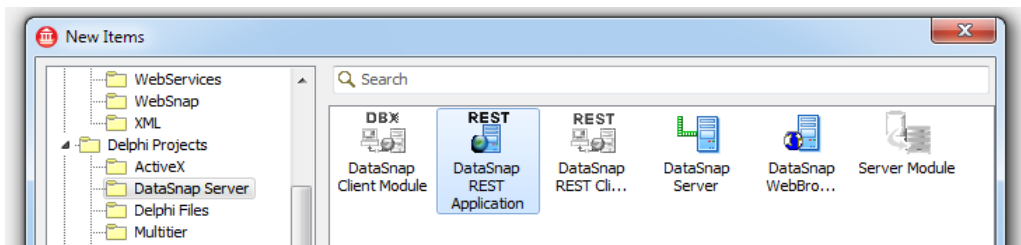
The remaining code outputs the entire object (getting back the original JSON representation, if nothing went wrong), the last (and only) name/value pair and frees the *TJSONObject* object (again, it is easy to cause memory leaks with this type of code):

```
Log (jsonObj.ToString);
Log (jsonObj.Get (jsonObj.Size - 1).ToString);
jsonObj.Free;
```

## THE DATASNAP REST WIZARD IN DELPHI XE

Now that I have introduced some of the key concepts related with REST and JSON in Delphi, it is time to start looking in practice into the REST support offered by DataSnap in Delphi XE. (Notice that DataSnap support is available only in the Enterprise and Architect editions of the product.)

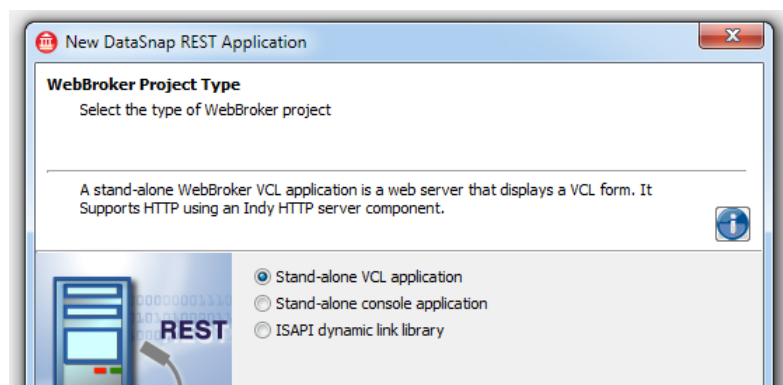
Our starting point in writing a REST server will be the new DataSnap REST Wizard. As an alternative, we could have created a more general DataSnap server application with HTTP support, which is another option to obtain a REST server. However, only by using the first approach, do we obtain an application with JavaScript support and ready-to-use HTML templates. As we'll see, this extended support provides a very good starting point for turning our DataSnap REST server into a full-blown web application. Below you can see the various DataSnap Wizards of Delphi XE:



You can see there are three different types of servers, a plain DataSnap Server, a DataSnap WebBroker Application, and a DataSnap REST Application. I'm going to demonstrate this last option.

## RUNNING THE WIZARD

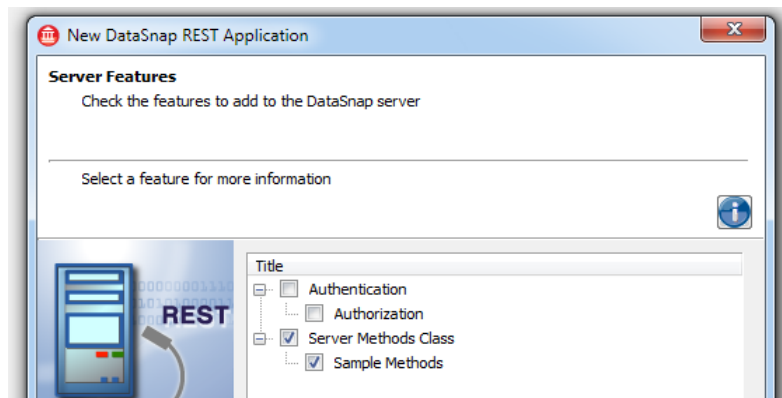
To build our first simple REST server in Delphi XE, we can use the DataSnap REST Application. As the first page of the wizard states (see image below), you can pick multiple alternatives all based on the WebBroker architecture, more or less like in the DataSnap WebBroker Application.



The two "stand-alone" options use the new WebBroker integration with the IdHTTPServer component, so that the application will effectively be a custom Web server. The third option is ISAPI. Potentially, though, you could also use other WebBroker options like the (now deprecated) Web App Debugger and the Apache Module support.

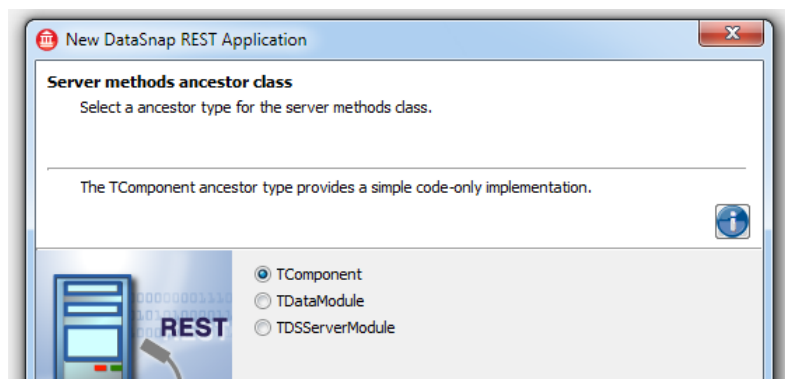
For this first project I've selected the "Stand-alone VCL application" option, which is best suited for testing and debugging. Since you are building a stand-alone web server, next you should select a port, and test its availability. The default suggested by the wizard is 8080, but any available port on your machine will do. This is just a property of a server and it is trivial to change it later on.

The third step is to select some of the features of the server, including authentication and authorization support:



Don't worry if you don't enable authentication now, as you can easily add it to the project later on. On the other hand, it is generally a good idea to add a server class with some sample methods the first time you use this wizard. The server class is the class exposing its methods as REST service requests, and you can add one (or more than one) later in the project.

Next, if you select a sample server class, the Wizard will ask you which base class to use. The available options are a plain *TComponent*, a *TDataModule* container (in which you can easily add non-visual components such as data access components), and a DataSnap server module. This last option makes little sense for a DataSnap REST application, but it is an interesting option for other types of DataSnap servers:



For this first project, I've chosen *TComponent* as the base class. In the last page of the wizard, you can select the base folder for the project. While you are prompted to enter the path of a folder, the folder name (the last element of the path you provide) will also double as project name. Since a project name must also be a valid identifier, the folder name cannot contain spaces and cannot start with a number.

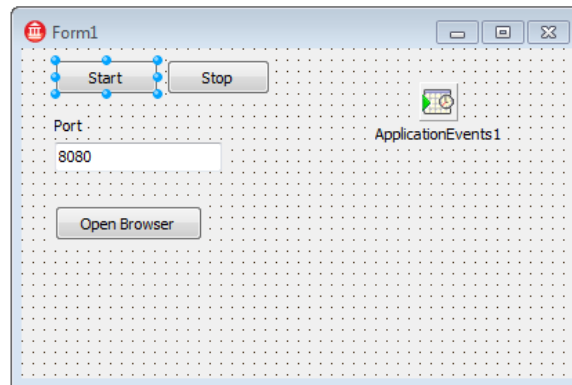
We need to specify a folder because the wizard generates a lot of support files in addition to the Delphi project source files. These support files include JavaScript files, Cascading Style Sheets, HTML templates, and even a few images.

## THE CODE GENERATED BY THE WIZARD

With the settings above, Delphi will generate a project with three units. If you want to see their source code, either follow the same steps in the wizard or look at the source code of the `RestXeOne` included in the source code bundled to this white paper.

- A main form with limited management options. This form will be removed as you change the project structure to an ISAPI DLL.
- A web data module (inheriting from `TWebModule`) hosting the DataSnap server components, several other support components, and the core of the WebBroker application.
- A unit with the server class, in which you add the methods of your REST server.

The main form has an edit box to easily change the server port at runtime, the start and stop buttons, and a button for starting the server and opening its default page in the your browser. This is the form at design time, before I made any change:



The Application Events component is used to update the user interface depending on the status of the server, by handling the *OnIdle* event. There is another very important element to this form, which is a *TIdHTTPWebBrokerBridge* object saved in a private field (*FServer*). This is a class that inherits from *TIdCustomHTTPServer* and supports the integration of this internal web server with the WebBroker architecture. The other two units of the application, in fact, are totally neutral to the architecture and will be identical for an ISAPI module or any other WebBroker application.

## THE WEB DATA MODULE GENERATED BY THE WIZARD

Let's now look at these web data module generated by the DataSnap REST Wizard in detail, before we add some code to the program and test it. A web module is the core element of the WebBroker architecture. It can define multiple actions tied to pathnames, can intercept other pathnames with *dispatcher* components, and might also do pre-processing and post-processing operations on any HTTP request.

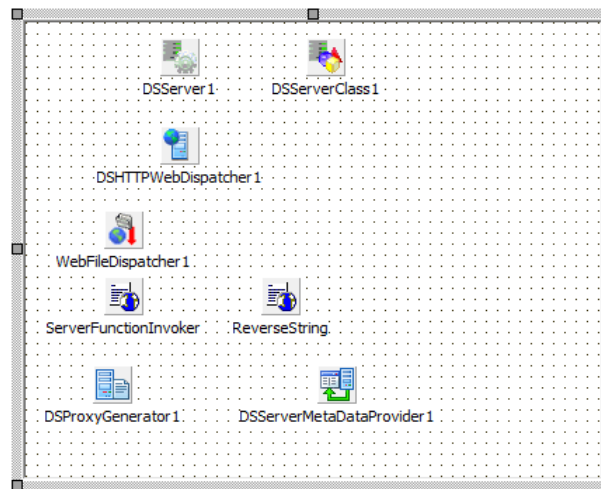
Here is a table with a summary of the pathnames and extensions the server will handle, the component or action managing them, and the source of the information:

Path or Extension	Component or Action	Target
/	DefaultAction	WebModuleDefaultAction method
/ServerFunctionInvoker	ServerFunctionInvokerAction	ServerFunctionInvoker page producer
/ReverseString	ReverseStringAction	ReverseString page producer
/datasnap*	DSHTTPWebDispatcher1	DataSnap processing
/datasnap/rest	DSHTTPWebDispatcher1	DataSnap REST processing
*.css, *.js, *.png	WebFileDispatcher1	Folders of the file system

The default action has a very limited role (it will redirect the request or return the output of the ReverseString page producer). This producer, also tied to its own pathname and action, returns a sample HTML page will look into later on.

In this web module, the core role is played by the *DSHTTPWebDispatcher* component, which is the entry point of the DataSnap and REST requests. This component intercepts any request with a URL starting with 'datasnap' (by default), which are passed to the DataSnap server component (DSServer) also included in the web module. For requests starting with 'datasnap' and indicating a 'rest' path (again a default), the processing will be diverted to the built-in REST engine. In other words, the requests starting with a 'datasnap/rest' path are considered as REST requests.

The complete set of component that the Wizard adds to this web module is visible in the image below, depicting this module at design time:



Here is a summary of the role of the various components:

- DSServer1 is the core DataSnap server component, gluing together all the various components of this architecture. Despite the fact it has only a couple of properties, its role is central.
- DSServerClass1 is the component that refers to the class exposing its methods as REST services, creates and manages the lifetime of objects of this class. Its role parallels that of a class factory in COM. The actual class this Server Class refers to is returned in the *OnGetClass* event handler of the component, in a reference parameter.
- DSHTTPWebDispatcher1 is the bridge between the WebBroker architecture and the DataSnap server, as already described.
- WebFileDispatcher1 is a WebBroker component (new in Delphi XE) used to return some of the support files (JavaScript files, images, Cascading Style Sheets) generated by the Wizards, and others you can add. A pre-processing event handler of this component will update the *serverfunctions.js* file every time you recompile and re-execute the application. This JavaScript file contains the “proxy” interface to the methods exposed by the REST server. The file is updated using the support provided by the next two components.
- DSProxyGenerator1 is a new Delphi XE component that you can use to generate the proxy interfaces for various programming languages (Delphi, C++, JavaScript). These are helper classes you can use in those languages to invoke the methods of the REST server easily, without having to manually create the calling URL. In this case the *Writer* property of the component is set to *'Java Script REST'*. This component works in conjunction with the next one.
- DSServermetaDataProvider1 offers to the DSProxyGenerator1 component the core metadata of the server. This is a support component with basically no specific settings.

- ReverseString is a page producer referring to the *ReverseString.html* file in the templates. This file provides a sample implementation of the user interface and application you can write in HTML and JavaScript to have a complete web user interface for one of your server methods. This will be the starting point for most of our examination. More details of this file are in a following section.
- ServerFunctionInvoker is a second page producer this time hosting a generic user interface generated from the metadata. The page produced will list all of the methods of the server and let you enter parameters and see the result in JSON format. This page is meant as a development and debugging tool, not for the end user (in fact it will be displayed only if invoked locally).

There are many other technical details and configuration option for these components, but given the scope of this paper, this description should be enough. Before we get to examine the actual effect of running this server, there are two more elements we have to consider: the code of the server class whose methods we are exposing and the role of the HTML and JavaScript user interface (which we'll investigate later).

## THE SAMPLE SERVER CLASS GENERATED BY THE WIZARD

The third unit that is generated by the DataSnap REST Application 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 *OnGetClass* event handler mentioned earlier, and is where most of the actual code will end up.

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

```
type
{$METHODINFO ON}
TServerMethods1 = class(TComponent)
private
    { Private declarations }
public
    { Public declarations }
    function EchoString(Value: string): string;
    function ReverseString(Value: string): string;
end;
{$METHODINFO OFF}
```

Notice that this class inherits from a plain *TComponent* base class and enables a special compiler option which produces extra RTTI information for public methods (predating Delphi 2010 extended RTTI): *{ \$MethodInfo ON }*.

The *EchoString* and *ReverseString* method will simply return the parameter you are passing or its reverse. We'll see which data types you can pass and return in a later section.

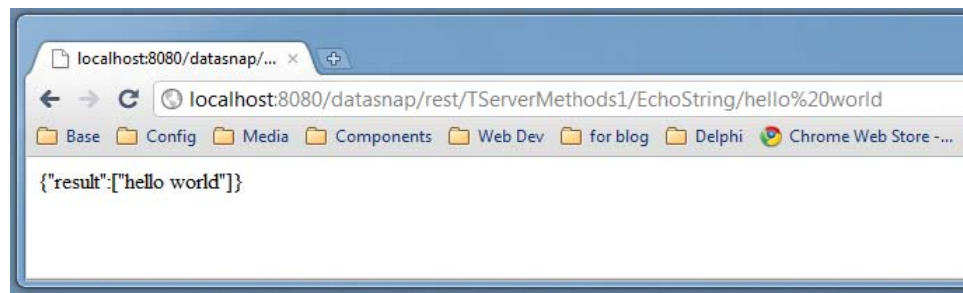
## COMPILING AND TESTING THE REST SERVER

We can now compile the server and see if it works. After compiling and running the program, you can use its simple user interface to turn on the server, using the *Start* button. Later on, we'll open the default web user interface (with the *Open Browser* button), but before we look at this, I want to explain the basics of how REST invocation works.

We can perform a test by opening a Web browser and manually enter the URL of one of the server methods, which is made of the application location address and port (*localhost:8080*), the REST path (*/datasnap/rest*) the name of the class (*TServerMethods1*), the name of the method you want to execute (for example, *EchoString*), and the (optional) parameters of the method:

`localhost:8080/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:

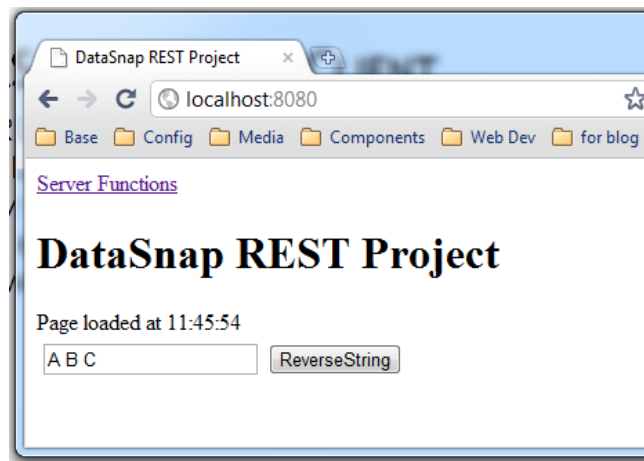


The JSON data returned by the server is an object with a "result" field containing an array of values, including the function result and any parameter passed by reference (it is an array because you can return multiple values).

## THE HTML AND JAVASCRIPT SAMPLE CLIENT

Of course, typing such a URL in the browser would be a total nonsense, but this is the starting point for building a browser based application. In fact, a sample of such an application is generated by the DataSnap REST Application Wizard in terms of HTML and JavaScript code. If we execute, in a browser, the default action of the custom web server, we'll see something like the following:





This is the HTML page with the skeleton stored in the *ReverseString.html* file and returned by the ReverseString page of the application's web module. Press the button and you'll actually see the string reversed. The HTML of the page (if we ignore the extra code used to manage login and described later) is quite simple:

```
<#serverfunctioninvoker>
<h1>DataSnap REST Project</h1>

<div>Page loaded at <span id="timeElement"></span></div>
<div id="contentdiv" class="contentdiv">
  <table>
    <tr>
      <td>
        <input id="valueField" type="text" value="A B C" />
      </td>
      <td>
        <button onclick='javascript: onReverseStringClick(); '>
          ReverseString</button>
        </td>
    </tr>
  </table>
</div>
```

The special tags with the # prefix are processed by the PageProducer's *OnTag* event handler, and are dynamically replaced by the Delphi application every time you ask for the page. These tags include element like the class to invoke, the availability of the "server function invoker" test page, and a few others:

```
procedure TWebModule2.ServerFunctionInvokerHTMLTag(
  Sender: TObject; Tag: TTag; const TagString: string;
  TagParams: TStringList; var ReplaceText: string);
```

```

begin
  if SameText(TagString, 'urlpath') then
    ReplaceText := string(Request.InternalScriptName)
  else if SameText(TagString, 'port') then
    ReplaceText := IntToStr(Request.ServerPort)
  else if SameText(TagString, 'host') then
    ReplaceText := string(Request.Host)
  // and so on...

```

As you can imagine by reading the HTML code above, a key role is played by the call to the *onReverseStringClick* JavaScript function, which has the following code (part of the same HTML file):

```

function onReverseStringClick()
{
  var valueField = document.getElementById('valueField');
  var s = serverMethods().ReverseString(valueField.value);
  valueField.value = s.result;
}

```

This function stores a reference to the input field, passes the value of this input field to the *ReverseString* method call, and updates the field value with the result. The *serverMethods* function returns an instance of the JavaScript proxy object defined in the *ServerFunction.js* source code file (the file that gets updated by the *DSProxiGenerator* component every time you recompile and re-execute the server).

The proxy object makes the Delphi server methods easily available to the JavaScript browser-based application, by encapsulating the HTTP request in an asynchronous call (using the AJAX technique):

```

function TServerMethods1(connectonInfo)
...
this.ReverseString = function(Value) {
  var returnObject = this.executor.executeMethod(
    'ReverseString', "GET", [Value], arguments[1],
    true, arguments[2], arguments[3]);
  if (arguments[1] == null) {
    if (returnObject != null && returnObject.result != null
        && isArray(returnObject.result)) {
      var resultArray = returnObject.result;
      var resultObject = new Object();
      resultObject.Value = Value;
      resultObject.result = resultArray[0];
      return resultObject;
    }
  }
}

```

```
        return returnObject;  
    }  
};
```

This is JavaScript code generated by Delphi from the RTTI of the classes exposed by the REST server. The JavaScript method manages the parameter and the return value, passing the first in the URL and extracting the second from the JSON object the REST function returns. This makes it relatively simple to call the REST server functions from the JavaScript browser application.

There are many other JavaScript support files that are used for the call, providing AJAX support, JSON encoding and much more, but it will be very complex to delve into them and that is beyond the scope of this paper.

## CALLING THE REST SERVER FROM A DELPHI CLIENT

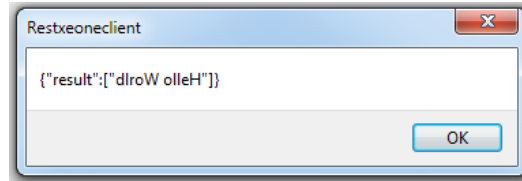
Now that we have built the server and we have seen how the server application can also expose a web application to a browser client, we can also write a Delphi client application for the same server. We can use two different approaches.

The first approach is to write a Delphi DataSnap client, using the specific REST client support introduced in Delphi XE. The second approach is to create a custom REST client and use the IdHTTP component to call the server and the DBXJSON support unit to parse the result. In the RestXeOneClient demo, I'll use both, starting with the "manual" one and later getting to the automatic support, which should be generally preferred.

First, I've created a standard client VCL application and added to its main form a button, an edit box, and an IdHTTP client component. In the button's *OnClick* event handler I've written the following code:

```
const  
    strServerUrl = 'http://localhost:8080';  
    strMethodUrl = '/datasnap/rest/TServerMethods1/ReverseString/';  
  
procedure TRestClientForm.btnManualClick(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 within the REST server, and the only parameter. If you enter the text "Hello World" in the edit box, 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, taking advantage of the JSON support available in Delphi and specifically discussed in section "Parsing JSON in Delphi".

As we have seen, 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, as in the final version of the method:

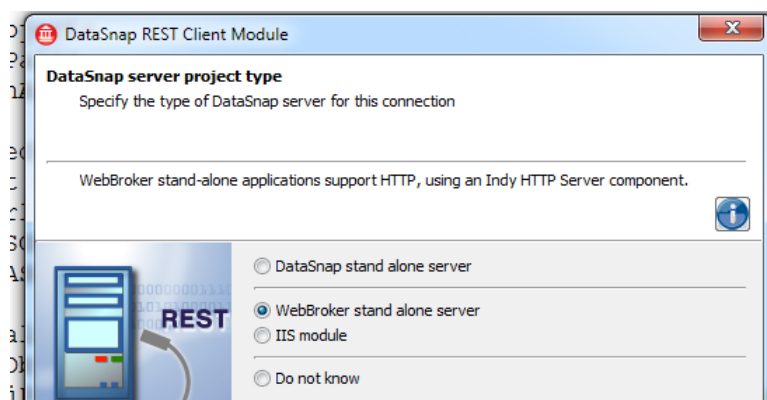
```
procedure TRestClientForm.btnManualClick(Sender: TObject);
var
    strParam, strHttpRequest, strResult: string;
    jValue: TJSONValue;
    jObj: TJSONObject;
    jPair: TJSONPair;
    jArray: TJSONArray;
begin
    strParam := edInput.Text;
    strHttpRequest := IdHTTP1.Get(strServerUrl +
        strMethodUrl + strParam);
    jValue := TJSONObject.ParseJSONValue(
        TEncoding.ASCII.GetBytes(strHttpRequest), 0);
    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
        edInput.Text := strResult;
    finally
        jValue.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.

## USING THE REST CLIENT SUPPORT

To make it much easier to write the client code, and still rely on the plain REST interface (and not the DataSnap HTTP support), we can use the new DataSnap REST Client Module Wizard, introduced in Delphi XE. Before you run this Wizard, make sure the server application is running and it not being executed in the Delphi debugger. You can obtain this using the Run without Debugging command of the Delphi IDE.

As you run the wizard, in the first page it will ask you if the server is local or remote (and you'd probably choose local) and in the second page which kind of architecture DataSnap the server is based on:



For this example, you should choose “WebBroker stand alone server” as I’ve done above. In the third page the wizard will ask you the server port, the DataSnap context (by default “/datasnap”), and optional account information. As you press Finish, the wizard will add two different units to your project:

- A client module unit with a data module hosting the DSRestConnection component, with the REST connection configuration.
- A unit with the proxy classes for calling the server, different from the client classes generated for a traditional DataSnap connection in that they use the DSRestConnection rather than the DBXConnection for invoking the server.

Notice that you don’t need to manually create the proxy object, because this is done automatically by the client module as you access a specific property added by the wizard to refer to the proxy itself:

```
// property ServerMethods1Client: TServerMethods1Client  
function TClientModule1.
```

```
GetServerMethods1Client: TServerMethods1Client;  
begin  
  if FServerMethods1Client = nil then  
    FServerMethods1Client := TServerMethods1Client.Create(  
      DSRestConnection1, FInstanceOwner);  
  Result := FServerMethods1Client;  
end;
```

This means we can invoke the server simply by referring to the client module's property and invoking the method itself. Needless to say this version of the code is much simpler than the direct call with manual JSON parsing that I wrote earlier:

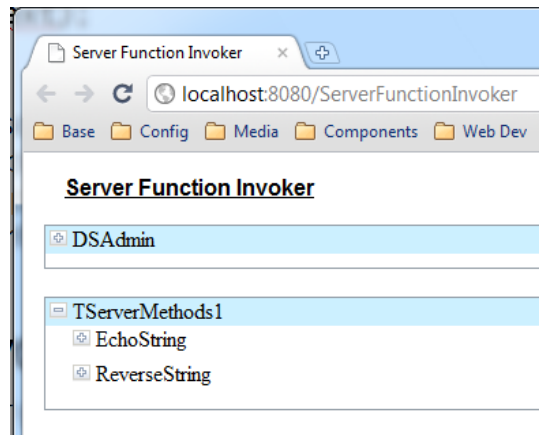
```
procedure TRestClientForm.btnProxyClick(Sender: TObject);  
begin  
  edInput.Text := ClientModule1.ServerMethods1Client.  
    ReverseString(edInput.Text);  
end;
```

The two techniques produce the same exact result. The key concept to keep in mind here is that as you develop a Delphi DataSnap REST Server, you are building a web application, enabling Delphi client to call it, and also making it possible to call the server with any other client, written in any language, as long as it can make an HTTP call and parse the JSON result.

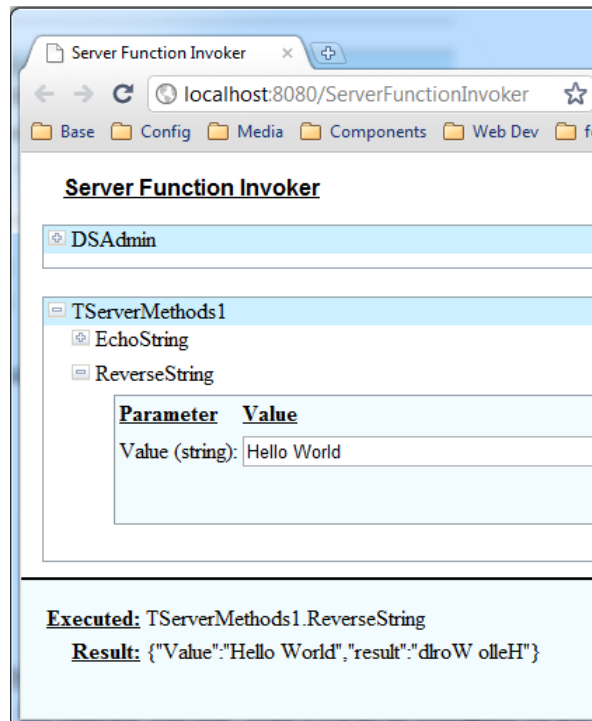
## THE SERVER FUNCTION INVOKER

There is one last feature of the server generated by the DataSnap REST Application Wizard that I haven't discussed so far. This is the server function invoker, a generic mechanism to test the various functions that the server makes available. This feature is supported through the second page producer of the web module, a corresponding HTML template, and a few JavaScript files.

Simply enter the path *"/ServerFunctionInvoker"*, or follow the link from the ReverseString page, and you'll get a HTML page with two areas, corresponding to the classes exposed by the server: the administrative class DSAdmin plus the specific classes of the application, in this case only TServerMethods1. Click on the plus sign on the side of the class and the page will expand the list of the methods:



If you now select a method, the JavaScript code behind the page will generate as many input fields as there are parameters in the method. You can type a value, press a button on the side, and see the plain JSON response:



This page gets automatically updated as you modify or extend the target server class, or add further classes to the server, and can be very handy for debugging and testing.

## AUTHENTICATION AND AUTHORIZATION

Now that we have seen the basic features of DataSnap REST application, we can look into a specific feature that is part of the wizard's generated code (even in case you omit the option): Authentication support.

Authentication management is tied to two other related topics. The first is session management, that is figuring out how to determine if different (stateless) HTTP requests come from the same client application. The second is authorization support, that is determining which user is allowed to call which server side method. All three topics are the subject of this section.

I'll start by creating another DataSnap REST Application, this time enabling the authentication and authorization options. In the resulting application, the web module will have one extra component, *DSAuthenticationManager*. This is referenced by the *AuthenticationManager* property of the *DSHTTPWebDispatcher* component.

## SESSIONS

Before we look into the authorization support though, I have to introduce session management in DataSnap. The first time a client calls the server, it creates a new session ID and returns it in a custom header (*'dsession'*). Subsequent requests from the client should include a *'Pragma'* header with the session ID, so that the server recognizes the request comes from the same client.

A session lasts until it expires (by default 1200 seconds, that is, 20 minutes) or it is forced to close using the special *CloseSession* URL. If you pass a session that doesn't exist (for example, because in the meantime you closed and reopened the server) you'll receive an exception and the session ID is reset.

Within the server-side code, you can access the current session using the call:

```
var  
    session: TDSSession;  
begin  
    session := TDSSessionManager.GetThreadSession;
```

You can also attach name-value pairs to the session data, using string values and the four methods *PutData*, *HasData*, *GetData*, and *RemoveData*.

When using a JavaScript client, the support code generated by the wizard manages the session ID with a global JavaScript variable (*\$\$\$SessionID\$\$\$*) and passes it to each following AJAX call in the same page. If you change the page or refresh it the session ID gets lost.



You can, however, force the JavaScript support code to add this session ID to a browser cookie, so that it remains available across pages. To do this, you have to add the following call in the *onLoad()* JavaScript function of the each HTML file:

```
ini tSessi onData(true);
```

A second change you might want to do is to change way the *loginRequired* variable is managed, but we'll get to this extra line of code as we look into authentication.

## AUTHENTICATION

The *DSAuthenticationManager* component has two key events, *OnUserAuthenticate* and *OnUserAuthorize*. The first is called every time the server receives a request from a non-authenticated user. After the user and its session are authenticated, the method is not called again. This is how you can implement the method, if you are not looking for real security:

```
procedure TWebModule2.DSAuthenti cationManager1UserAuthenti cate(
  Sender: TOb ject; const Protocol, Context, User, Password: string;
  var valid: Boolean; UserRoles: TString s);
var
  sessi on: TDSSessi on;
begin
  valid := (user = password) and (user <> '');

  if valid then
    begin
      Sessi on := TDSSessi onManager. GetThreadSessi on;
      Sessi on. PutData ('username', user);
```

In general, you'll look-up in a database or configuration file or operating system mechanism to validate the users and their passwords. In this case I'm accepting any non-empty username matching the password. I'm also adding the user name as an extra session parameter, something not terribly useful as this information is already available in the username property of the session.

As an aside, I've added an extra sever function, *GetUserName*, which returns both the session user name and the extra session parameter I created above:

```
function TServerMethods1.GetUserName: string;
var
  Sessi on: TDSSessi on;
begin
  Sessi on := TDSSessi onManager. GetThreadSessi on;
  Result := Sessi on. GetData ('username') + ':' + Sessi on. Username;
```

end;

Session information aside, if you now try to invoke a REST URL directly from the browser, you'll receive a standard HTTP login request. The JavaScript user interface, instead, will be informed by the server that there is an attached `DSAuthenticationManager` component, using the `<#loginrequired>` field that is references in the JavaScript startup code:

```
var loginRequired;
function onLoad()
{
```

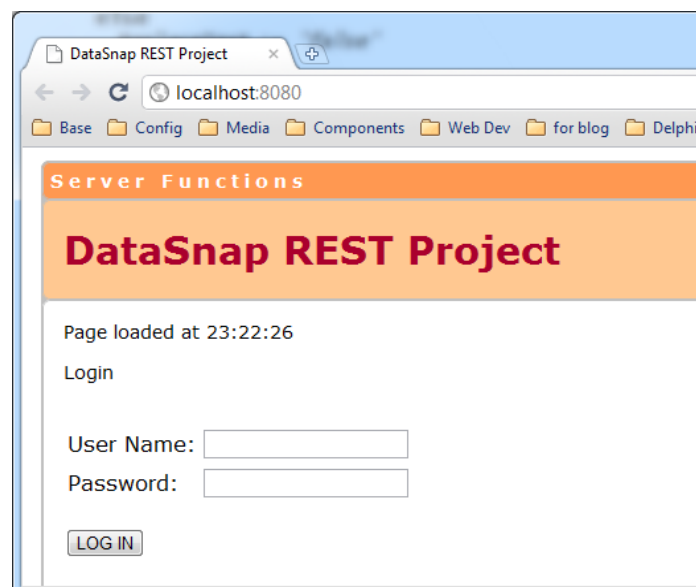
```
    loginRequired = <#loginRequired>;
```

Now on the server this value is process by the following code:

```
else if SameText(TagString, 'loginrequired') then
    if DSHTTPWebDispatcher1.AuthenticationManager <> nil then
        ReplaceText := 'true'
    else
        ReplaceText := 'false'
```

When the `loginRequired` variable is enabled, the browser-based application hides the standard user interface and displays a login form included in the standard HTML template. This takes place in the `ShowLogin` function by hiding and showing one of the two portions of the HTML page, the `logindiv` and the `contentdiv`.

The first time you open the main page of the web server, you'll get a login prompt rather than the expected user interface (notice that from this example onwards I've applied a simple Cascading Style Sheet to the demos, to make them look somewhat nicer):



Login with the user name and the password, and you'll see the standard interface. Now if you refresh the page, the session information is lost and you'll need to login again. As I mentioned earlier, there is a way to fix the session management using a cookie, but you'll still get a login prompt because the login request depends on the presence of the Authentication Manager, rather than on the current status of the session. This can be fixed by replacing the initial JavaScript code with a test on the local session value:

```
var LoginRequired = false;

function onLoad()
{
    initSessionData(true);
    showTime();
    LoginRequired = (getSessionID() == null);
}
```

In other words, the login is required only if the client application doesn't have a valid session ID (the *getSessionID()* function also checks if the session ID is expired).

## ROLES AND AUTHORIZATION

Once the server has authenticated the client application, it can execute any of the methods exposed by the server. You can filter the available methods depending on the user by handling the *OnUserAuthorize* event of the AuthenticationManager component. This is powerful but quite tedious. Notice that if you decide to avoid using the *OnUserAuthorize* event you have to remove the line generated by the wizard that sets the *valid* reference parameter to True.

As a simpler alternative, the DataSnap architecture has built-in support for user roles. As you authenticate a user, you can assign one or more roles to him or her. For this purpose the *OnUserAuthenticate* event handler of the example has this extra code:

```
if valid then
begin
    UserRoles.Add ('standard');
    if user = 'admin' then
        UserRoles.Add ('admin');
```

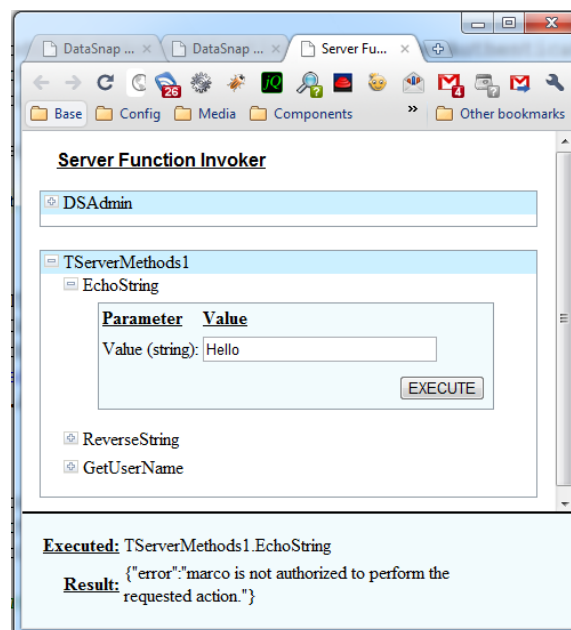
Each user is added to the standard role, but the administrator also has the admin role. As this is a string list, each user can be assigned to multiple roles. But how do we determine which method can be called depending on the user role? This is accomplished by attaching the *TRoleAuth* custom attributes to the server class as a whole or some of its methods (Custom attributes is a language feature first introduced in Delphi 2010, and covered, among other sources, in the help file and in my *Delphi 2010 Handbook*):

**uses**`DSAuth;`**type**

```
{ $METHODINFO ON }  
TServerMethods1 = class(TComponent)  
public  
    [TRoleAuth ('admin')]  
    function EchoString(Value: string): string;  
    [TRoleAuth ('standard')]  
    function ReverseString(Value: string): string;  
    [TRoleAuth ('standard')]  
    function GetUserName: string;
```

It is very important to notice that if you fail to add the `DSAuth` unit before this class declaration, the compiler will not understand the custom attribute, will produce a vague warning (*Unsupported language feature: 'custom attribute'*) and the roles won't be enabled in your application. Consider also that the `TRoleAuth` attribute parameter can be a comma-separated list of roles, and that there is a second optional parameter with a list of denied roles you can use as an alternative.

With the specified settings, all users can call the `ReverseString` and `GetUserName` methods, while the `EchoString` method is reserved for administrators. If another user tries to call the `EchoString` method, the server will raise an exception which will be returned in the JSON response. This is an example of the error in the Server Function Invoker page in the browser:



## THE DATA TYPES OF A DATASNAP REST SERVER

Now that I've given you an overview of sessions, authentication, and authorization in DataSnap, it is time to move to a new topic. In this initial demo, server methods always returned strings. Obviously, you can use more native types like Integers, characters, and Booleans.

But what about returning more complex data types? DataSnap has support for passing objects, datasets, and streams, plus the ability to manage the JSON response directly. Also, some of these native structures have automatic support for filtering and caching. These are all topics I intend to cover in this section of the article. To demonstrate these features I've created a single large application with a server class returning many different data types.

*Notice that in this section I'll mostly focus on the server side code and the generated JSON data structures, while in a later section "jQuery and the Data Types Demo" I'll create an HTML client application showing how to call most of the methods of this server. The reason for this delay is I want to introduce the jQuery library first, and then use it for building the client side application.*

### ORDINAL TYPES

A DataSnap REST server can directly return native types, which are mapped to the corresponding JSON structures whenever possible. Here are a few functions returning basic types:

```
public
function AddInt (a, b: Integer): Integer;
function AddDouble (a, b: Double): Double;
function IsOK: Boolean;
function GetLargeString (nCount: Integer): string;
```

These are sample results for these four methods (the last one returns a string with the number of characters passed as parameter, with an x every ten elements – will see later on why):

```
Executed: TServerMethods1.AddInt
Result:   {"a": "1", "b": "2", "result": 3}
```

```
Executed: TServerMethods1.AddDouble
Result:   {"a": "1.5", "b": "4.4", "result": 5.9}
```

```
Executed: TServerMethods1.IsOK
Result:   {"result": true}
```

Executed: TServerMethods1.GetLargeString  
Result: {"nCount": "20", "result": "oooooooooooooooooooo"} }

In the first three cases, the server is returning the value in the native JSON format (Number or Boolean) and not a (quoted) string value. Other basic data types, such as sets or enumerated values, cannot be returned directly. The same is true for dates. Some user-defined data structures such as records are also not managed.

## OBJECTS AND MARSHALING

One very important feature is the ability to return an object from a DataSnap server, including a REST server. In Delphi 2010, you could accomplish this by explicitly using the *TJSONMarshal* (and *TJSONUnmarshal*) support classes, while in Delphi XE this operation takes place automatically.

Marshaling support is based on the use of RTTI for private fields. While Delphi developers generally think of the data of an object as the set of its published properties (as happens with components streaming and the DFM format), in this case the actual internal data of the objects is considered. Of course, not all of the data types of the fields are mentioned, or mentioned properly. As the comments in the *DBXJSONReflect* unit states:

*These are the fields for which there is already a built-in conversion/reversion: integer, string, char, enumeration, float, object. For these types the field values are ignored and user conversion is expected: set, method, variant, interface, pointer, dynArray, classRef, array.*

For other field types, you can enable a conversion (and reversion) either by enabling a ready-to-use conversion supplied in that unit or writing your own. The way you enable conversions is by using the *JSONReflect* attribute. Although I have no time to consider all options (and I'll later also explain why), here are a couple of examples.

First, consider a simple object structure, like the following:

```
type
  TPerson = class
  private
    FAge: Integer;
    FLastName: string;
    FFirstName: string;
  procedure SetAge(const Value: Integer);
  procedure SetFirstName(const Value: string);
  procedure SetLastName(const Value: string);
  public
```

```

    property LastName: string read FLastName write SetLastName;
    property FirstName: string read FFirstName write SetFirstName;
    property Age: Integer read FAge write SetAge;
end;

```

This is a method returning such an object:

```

function TServerMethods1.GetPerson: TPerson;
begin
    Result := TPerson.Create;
    Result.LastName := 'Smith';
    Result.FirstName := 'Joe';
    Result.Age := 44;
end;

```

And this is the resulting JSON data structure:

```

Executed: TServerMethods1.GetPerson
Result: {
  "result": {
    "type": "ServerMethodsUnit1.TPerson",
    "id": 1,
    "fields": {
      "FAge": 44,
      "FLastName": "Smith",
      "FFirstName": "Joe"
    }
  }
}

```

Now suppose you return an object like a *TStringList*. Getting the internal set of fields is worthless because they don't include the actual data, which is referenced by a pointer:

```

function TServerMethods1.GetNames: TStringList;
begin
    Result := TStringList.Create;
    Result.Add('one');
    Result.Add('two');
    Result.Add('three');
end;

```

This is the *useless* JSON data you'll receive (useless because the actual content, the strings in the string list, isn't there):

```

Executed: TServerMethods1.GetNames
Result: {
  "result": {

```

```

"type": "Classes.TStringList",
"id": 1,
"fields": {
  "FCount": 3,
  "FCapacity": 4,
  "FSorted": false,
  "FDuplicates": "duplicate",
  "FCaseSensitive": false,
  "FOwnsObject": false,
  "FEncoding": null,
  "FDefaultEncoding": {omitted},
  "FDelimiter": "",
  "FLineBreak": "",
  "FQuoteChar": "",
  "FNameValueSeparator": "",
  "FStrictDelimiter": false,
  "FUpdateCount": 0,
  "FWriteBOM": true
}}

```

Suppose we have a *TStringList* field in a class, like the following extension of the *TPerson* class used earlier:

```

type
TPersonData = class (TPerson)
private
  FDateOfBirth: TDate;
  FMoreData: TStringList;
  procedure SetMoreData(const Value: TStringList);
  procedure SetDateOfBirth(const Value: TDate);
public
  constructor Create;
  destructor Destroy; override;
  property MoreData: TStringList read FMoreData write SetMoreData;
  property DateOfBirth: TDate
    read FDateOfBirth write SetDateOfBirth;
end;

```

Here we have two problems. The data will be returned in its numeric representation and the string list without its actual values, but only as shown above. The solution in this case is to either enable (in case of the *TStringList* field) or write (in case of the *TDate* field) conversion methods. This is done in Delphi XE by using the *JSONReflect* attribute that I mentioned earlier. This attribute has a number of field and somewhat limited documentation, so it is not really easy to use, but again I don't want to delve too much into this topic.



Here is the class declaration with the two attributes. The first *JSONReflect* maps the date to a string, using a *TISODateTimeInterceptor* support class that I found in the (very interesting) *ServerProjectWithUserTypes* demo, part of the official set of Delphi XE DataSnap demos. The *JSONReflect* maps the string list to an object, with custom data:

```

type
  TPersonData = class (TPerson)
  private
    [JSONReflect(ctString, rtString,
      TISODateTimeInterceptor, nil, true)]
    FDateOfBirth: TDate;

    [JSONReflect(ctTypeObject, rtTypeObject,
      TStringListInterceptor, nil, true)]
    FMoreData: TStringList;

    procedure SetMoreData(const Value: TStringList);
    procedure SetDateOfBirth(const Value: TDate);
  public
    ... as above

```

Now if you write a method that returns an object on this class, with proper data filled in, you'll get JSON with proper values for the data and the actual strings of the string list:

```

Executed: TServerMethods1.GetPersonWithData
Result: {
  "result": {
    "type": "ServerMethodsUnit1.TPersonData",
    "id": 1,
    "fields": {
      "FDateOfBirth": "1899-12-30 00:00:00",
      "FMoreData": {
        "type": "DBXJSONReflect.TSerStringList",
        "id": 2,
        "fields": {
          "FSerStringItemList": [
            {"type": "DBXJSONReflect.TSerStringItem",
              "id": 3,
              "fields": {"FString": "Hello", "FObject": null}},
            {"type": "DBXJSONReflect.TSerStringItem",
              "id": 4,
              "fields": {"FString": "Joe", "FObject": null}}
          ],
          "FSorted": false,
          "FDuplicates": "duplignore",

```

```

        "FCaseSensitive": false
    }},
    "FAge": 44,
    "FLastName": "Smith",
    "FFirstName": "Joe"
}}}
```

The internal data structure is not simple, but we expose a lot of internal information about the string list (and potentially also the objects hooked to it). Now, given this is quite powerful, you might be wondering why I'm sort of downplaying this approach. As much as I find this interesting, you should consider that objects to JSON marshaling makes sense mostly in a Delphi-to-Delphi communication layer. Since my focus here, instead, is to use the browser as a client, an object representation based on internal Delphi class names and data structures is of limited use. JavaScript certainly won't need them.

The alternative, as we'll see later, is to return a *TJSONObject* directly, with better and direct management from the JavaScript side. This is the approach I prefer and will cover after we finish examining data types for which Delphi has specific native support.

## DATASETS AND STREAMS

The first relevant complex data type that a DataSnap REST server can handle automatically is a dataset. You can return as object of any class that descends from *TDataSet*, and get it properly converted to JSON.

As an example, consider the following server side method, which returns a complete *ClientDataSet*:

```

function TServerMethods1.GetCustomers: TDataSet;
begin
    ClientDataSet1.Open;
    ClientDataSet1.MergeChangeLog; // clean up, just in case
    Result := ClientDataSet1;
end;
```

The component has a trivial configuration, loading the data from a file in the same folder of the program:

```

object ClientDataSet1: TClientDataSet
    FileName = 'customer.cds'
End
```

As you can see in the following excerpt of the resulting JSON, Delphi will return separate data and metadata, and will arrange the data by column (something I find quite unusual):

```

Executed: TServerMethods1.GetCustomers
Result: {"result":
  {"table": [
    ["CustNo", 7, 0, 0, 0, 8, 0, 0, false, false, 0, false, false],
    ["Company", 1, 1, 0, 30, 31, 0, 0, false, false, 0, false, false],
    ["Addr1", 1, 2, 0, 30, 31, 0, 0, false, false, 0, false, false]... ],
  "CustNo": [1221, 1231, 1351, 1354, ... ],
  "Company": ["Kauai Dive Shoppe", "Unisco",
    "Sight Diver Webinartist", "Cayman Divers World Unlimited", ... ],
  "Addr1": ["4-976 Sugarloaf Hwy", "PO Box Z-547", "1 Neptune Lane",
    "PO Box 541", ... ],
  ... }}

```

The *table* array has the field definitions, with types and other settings not terribly readable, while the following fields represent the columns, each with the complete set of values. As we'll see later it is possible to return only a range of values in the columns, starting with a given record number and returning a limited set. I'll focus on this extension in the section on filtering.

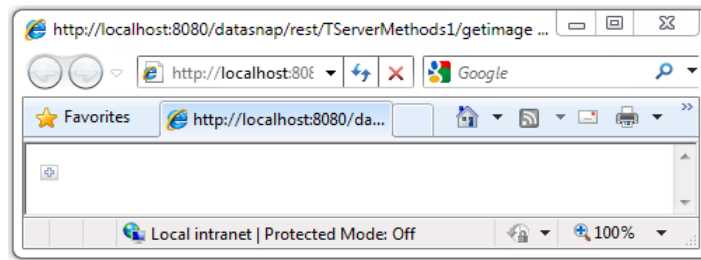
A second custom Delphi class that is managed natively by a DataSnap REST server is the *TStream* class (and its descendant classes). Consider the following server method:

```
function TServerMethods1.GetImage: TStream;
var
    fStream: TFileStream;
begin
    fStream := TFileStream.Create('images\expand.png', fmOpenRead);
    Result := fStream;
end;
```

If you use the Server Function Invoker client, you'll get a result like the following (this is only the initial portion):

Executed: TServerMethods1.GetImage  
Result: "%PNG\r\n\u001a\n\u0000\u0000\u0000\u0000\.

As you can see, this isn't JSON. When you return a binary stream, in fact, you'd actually want to get the actual data, not a textual representation. If you open the corresponding URL (<http://localhost:8080/datasnap/rest/TServerMethods1/getimage>) in Internet Explorer, you'll indeed get the image, as displayed below. However other browsers (such as Chrome) won't render it because of the lack of a proper Content-Type header in the HTTP response.



For streams, however, you can also ask for a conversion to JSON, adding to the URL a special token *json=true*. If you use the following URL, you'll get the response below:

`http://localhost:8080/datasnap/rest/TServerMethods1/getimage?json=true`

```
{"result": [[137, 80, 78, 71, 13, 10, 26, 10, 0, 0, 0, ... ]]}
```

Notice that returning the result in binary format is supported only if there are no other *var* parameters in the function call.

## RETURNING JSON DIRECTLY

A further data type that is directly supported by the DataSnap REST is the *TJSONValue* type, and its subclasses including *TJSONObject* and *TJSONArray*. As I already mentioned, this is a very interesting option if your goal is to use JavaScript for the development of the client application. For example, we can return the data for an object by creating a corresponding JSON object:

```
function TServerMethods1.GetPersonJson: TJSONObject;  
begin  
    Result := TJSONObject.Create;  
    Result.AddPair ('LastName', 'Smith');  
    Result.AddPair ('FirstName', 'Joe');  
    Result.AddPair ('Age', TJsonNumber.Create (44));  
end;
```

The response to this request is plain JSON data structure, much simpler to handle in a JavaScript client:

```
Executed: TServerMethods1.GetPersonJson  
Result:   {"result": {  
            "LastName": "Smith",  
            "FirstName": "Joe",  
            "Age": 44}}
```

Similarly to return a list of names (our original string list) it will be much simpler to use a JSON array directly, specifically for client-side processing, like I'm doing in this server function:

```
function TServerMethods1.GetNamesJson: TJSONArray;  
begin  
    Result := TJSONArray.Create;  
    Result.Add('one');  
    Result.Add('two');  
    Result.Add('three');  
end;
```

Again, the resulting JSON will be quite simple: Compare it with the structure of the *TStringList* property and you'll clearly see the difference:

```
Executed: TServerMethods1.GetNamesJson  
Result:   {"result": ["one", "two", "three"]}
```

In this example, I've focused on the data structures the server can return. In a later demo I'll build a complete client using the same JSON results, to clarify the entire picture.

## FILTERING AND CACHING

Before I get to the examples focused on the JavaScript client, there are two more relevant new features of the REST support in DataSnap introduced in Delphi XE that are worth mentioning, although I won't delve into them in too much detail.

The first is the ability to filter a result (namely a string or a dataset) using a specific filtering syntax that is available in a manual call (using the URL) or providing values for the *arguments* array used during the JavaScript proxy calls.

The two filters are substring (*ss*) and table (*t*), both with three available operations (count (*c*), offset (*o*), and range(*r*)). All of these elements are added as URL parameters, after the pathname of the server function and the function parameters (which are passed on the path).

In this example, we will get the first 20 characters out of a 100-character string (the server is the same as the previous example) or skip the first 85 characters passing an offset. Here are the two URLs and the corresponding JSON results:

```
/datasnap/rest/TServerMethods1/getlargestring/100?ss.c=20  
{"result": ["oooooooooooooooooooo"]}
```

```
/datasnap/rest/TServerMethods1/getlargestring/100?ss.o=85
```

```
{"result": ["0000x0000000000x"]}
```

The range filter takes two parameters, the offset and the count. For example, you can retrieve only the third record of a table using `"t.r=2,1"`, since the first value of the range is zero-based:

```
/datasnap/rest/TServerMethods1/GetCustomers?t.r=2,1
{"result": [{"table": [...], "CustNo": [1351], "Company": ["Sight Diver
Webinarist"], "Addr1": ["1 Neptune Lane"], "Addr2": [null], "City": ["Kato
Paphos"], "State": [null], "Zip": [null], "Country": ["Cyprus"], "Phone": ["357-
6-876708"], "FAX": ["357-6-870943"], "TaxRate": [0], "Contact": ["Phyllis
Spooner"], "LastInvoiceDate": ["1994-10-18 19:20:30.0"]}]}
```

Having a partial amount of data for every request would make it quite easy to paginate a large result set from the HTML client, returning, say, 20 records every time and creating multiple pages of data.

## CACHING DATA

When you are returning a stream or some other complex structures, it is nice to be able to return the full binary data rather than a JSON representation. But what if you need to return also some other data structures, or returns two different streams in a single call?

This is where DataSnap REST caching support might come handy. Every time you have a method returning multiple data types (using `var` or `out` parameters), you can ask for the data in a specific format rather than the JSON result.

Suppose you have this server-side method (part of the same `RestXEDataTypes` demo covered earlier):

```
function TServerMethods1.ImageAvailable(const strImageName: string;
out aStream: TStream): Boolean;
begin
  Result := False;
  if strImageName = 'expand.png' then
    begin
      aStream := TFileStream.Create('images\expand.png', fmOpenRead);
      Result := True;
    end;
end;
```

If you make a standard call, you'll receive the JSON representation of the function result plus the stream. (Remember, by default when you access to a stream you get the binary data, and you can generally opt to ask for the JSON format). In this case you can also opt

to ask for the *application/rest* format in the *Accept* header of the HTTP request. If you are using JavaScript for the call, rather than using the proxy you'll have to use a direct call to the core *ServerFunctionExecutor* object.

Here is a JavaScript code snippet with the call:

```
function onReverseStringClick()
{
    var valueField = document.getElementById('valueField');
    var methExec = new ServerFunctionExecutor("TServerMethods1");

    var resultObj = methExec.executeMethod("ImageAvailable", "GET",
        [valueField.value], null, true, null, "application/rest");
    document.getElementById('image1').src =
        methExec.getCacheURL(resultObj.result[0], true);
}
```

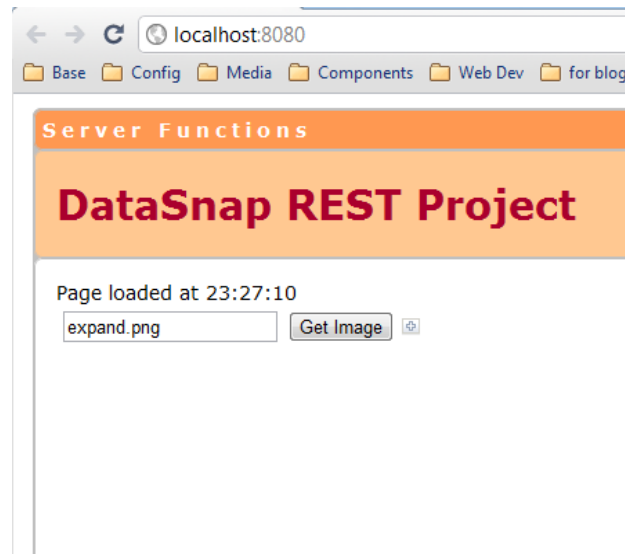
The parameters you pass to the generic *executeMethod* function of the *ServerFunctionExecutor* object are the method name, the request type, the parameters, the callback, if it has a result, the request filters, and the accept format. This call results in the following JSON:

```
{"result": ["0\0\0", true], "cached": 0, "cmdIndex": 0}
```

As you can see in the last JavaScript line above, you can ask for a GET request using the URL build from the cache, which is something like:

```
http://localhost:8080/datasnap/cache/0/0/0?sid=<...>
```

This will return the binary data of the stream, without processing the server method again, but returning the object that was created earlier, not returned, but kept in the cache. So again, in last JavaScript line above we can assign this URL of the cached stream to an image element of the page, and get to see the image:



## USING JQUERY

Delphi's DataSnap REST servers include a set of JavaScript support files, mostly devoted to simplify calling the servers themselves. In terms of DOM manipulation (the dynamic changes you can do to HTML pages in the browser), you are down to using plain old JavaScript. There are several JavaScript libraries you can use to simplify the development of browser applications based on this language. One of the best known and most actively used today (and backed by companies like Google and Microsoft) is jQuery. This is an incredibly powerful and open source JavaScript library available at:

<http://jquery.com>

I don't have time to delve into jQuery or to explain why I prefer this library over the many alternatives, but I want to provide a very short intro to jQuery showing how you can improve the standard HTML page generated by the Delphi DataSnap REST Wizard by using it... combining jQuery with the code Delphi provides. (If you look at the examples of the white paper I wrote last year, based on my Delphi 2010 Handbook, you'll see that I used jQuery exclusively, only because there was no JavaScript client support in Delphi back then.)

There are a few key concepts to get an idea of how jQuery works. First, jQuery is an object and you can wrap it (and its methods) around any existing JavaScript object or DOM element. To do this, you use the jQuery function (also references by the \$ shorthand) passing to it an object directly or a selector, a rule for selecting one of more objects based on its ID, class, or tag.



For example, given a web page, you can respectively hide all hyperlinks of a page, show all elements with a given class, and toggle the visibility of the element with the given ID by writing these three lines:

```
$("#a").hide()  
$(".myclass").show()  
$("#theid").toggle()
```

Another key concept in jQuery is that you don't attach behavior to HTML elements by tying the JavaScript to them (which makes the code harder to read and much more difficult to change by a web designer).

## ADDING JQUERY TO THE WIZARD'S HTML TEMPLATE

I know this is a very minimal description, but I cannot spend the couple dozen pages that a good introduction would require. For now, let us focus instead on how we can take the code generated by the wizard and improve it by embedding jQuery in the project. (This is demonstrated in the *RestXeOnejQuery* demo project.)

All of our changes, in this first example, will be in the *ReverseString.html* template. First we need to add a line of code to include jQuery (after adding the *jQuery.js* file to the *js* folder of the project):

```
<script type="text/javascript" src="js/jquery.js"></script>
```

As an alternative, you can refer to one of the public locations for jQuery, but this is slightly dangerous because something could change in the library and cause your code to stop working. Next, we can focus on the code of the *onReverseStringClick* function. The original text was:

```
function onReverseStringClick()  
{  
    var valueField = document.getElementById('valueField');  
    var s = serverMethods().ReverseString(valueField.value);  
    valueField.value = s.result;  
}
```

We can use jQuery to replace the *getElementById* calls and use the *val()* function to access the elements' value:

```
function onReverseStringClick()  
{  
    var valueField = $('#valueField');  
    var s = serverMethods().ReverseString(valueField.val());  
}
```

```

    valueField.val (s.result);
}

```

A second change is that the method was hooked to the HTML directly:

```

<table>
  <tr>
    <td><input id="valueField" type="text" value="A B C" /></td>
    <td><button onclick='javascript: onReverseStringClick();' >
      ReverseString</button></td>
  </tr>
</table>

```

By using jQuery we can remove the *onclick* attribute above, and add an identifier to it:

```

<td><button id="reverseButton">ReverseString</button></td>

```

Now the HTML has no reference to the code. This is very important because it makes this HTML much better suited for customization by a web designer, with limited knowledge of the application code and requirements.

The approach used by jQuery is to associate the two event handlers with the DOM element in code (which can be in a separate file from the HTML page) using the following initialization function:

```

$(document).ready(function() {
  $('#reverseButton').click(function () {
    onReverseStringClick();
  });
});

```

This code is executed as the document (the HTML page) becomes ready and installs a handler function for the click event of the given element, using an anonymous method (which in turns calls *onReverseStringClick*).

## JQUERY AND THE DATA TYPES DEMO

Now that we have seen how you can embed jQuery in the sample HTML template generated by the Delphi DataSnap REST Wizard, we can try to leverage this library in a more real world situation. I'll create a client (an HTML page with JavaScript code) for the various methods made available by the *ServerProjectWithUserTypes* demo project I covered in the section "The Data Types of a DataSnap REST Server".

To support this new user interface, I've added to the web module of the project a new action (*/jclient*) and a corresponding Page Producer component connected with a new

HTML template, *JClient.html*. The initial part of the page is visible below (with the result of some of the requests):

Since this HTML page has quite a few forms for the various methods, I've adopted a specific structure. I've enclosed each form in an external *div* named after the function and referring to the forms elements with the combined ids. As an example, this is the form for the *AddInt* operation:

```
<h3>function AddInt (a, b: Integer): Integer</h3>
<div id="AddInt">
  <table>
    <tr>
      <td><label>a: </label>
        <input id="a" type="text" value="10" /></td>
      <td><label>b: </label>
        <input id="b" type="text" value="32" /></td>
      <td><label>result: </label>
        <input id="result" type="text" value="" /></td>
      <td><button id="button">AddInt</button></td>
    </tr>
  </table>
</div>
```

The page initialization code (`$(document).ready`) installs a handler for the button, referring to the various input controls:

```
$('#AddInt #button').click (function () {  
    res = serverMethods().AddInt(  
        $('#AddInt #a').val (),  
        $('#AddInt #b').val ());  
    $('#AddInt #result').val (res.result);  
});
```

All of the forms and methods follow the same approach, and you can look some of them up in the companion code, like the *AddDouble* or *GetLargeString* forms. The form for the *IsOK* function is slightly different in that it has no parameters and uses the returned value to change the color of the text:

```
$('#IsOK #button').click (function () {  
    res = serverMethods().IsOK ();  
    if (res.result)  
    {  
        $('#IsOK #result').val ('ok');  
        $('#IsOK #result').attr('style', 'color: green;');  
    }  
});
```

For testing the *GetLargeString* function, I added several buttons as we might want to ask the server to filter the resulting JSON, for example, returning only the first 20 characters. Since we need to pass an extra HTTP field, we need to make a low-level call using the *ServerFunctionExecutor* object directly, and pass as second last parameter the HTTP fields in a JSON object:

```
$('#GetLargeString #button2').click (function () {  
    var methExec = serverMethods().executor;  
    res = methExec.executeMethod("GetLargeString", "GET", [100],  
        null, true, {"ss.c": "20"}, null);  
    $('#GetLargeString #result').val (res.result);  
});
```

Accessing an object marshaled by the server is quite simple, as the only extra element is to refer to the *fields* field:

```
res = serverMethods().GetPerson();  
var obj = res.result.fields;
```

Things get a little out of control if we want to access the strings passed in the string lists parameter:

```
var stringList =
  res.result.fields.FMoreData.fields.FSerStringItemList;
$('#GetPersonWithData #list').val (
  stringList[0].fields.FString + ' ' +
  stringList[1].fields.FString);
```

This code is really ugly, and a good reason to avoid using Delphi automatic marshaling in complex situations.

A data structure that is a little more complex, but can certainly be managed from a client, is the JSON format returned by Delphi for a *TDataSet*. As we saw, there is a first field called "table" and containing metadata (an array for fields and their settings), which we can parse to create a table header dynamically:

```
res = serverMethods().GetCustomers().result;
var strHtml = '<tr>';
for (var n=0; n < res.table.length; n++) {
  strHtml = strHtml + '<th>' + res.table[n][0] + '</th> ';
}
strHtml = strHtml + '</tr>'
$('#GetCustomers #result').html(strHtml);
```

In the JSON result, there is also a field for each column with the list of values. Although the fields are "by column", we can parse this array structure and extra the fields "by record". A way to accomplish this is to compute the number of records for one of the fields (*res.CustNo.length*), loop on the record numbers (with the *m* variable), loop again into the metadata (with the *n* variable), and find a matching field at the main object level:

```
fieldname = res.table[n][0];
fieldvalue = res[fieldname][m];
```

I know this sounds confusing, but I'm not sure that looking to the complete code will makes things easier to understand:

```
for (var m=0; m < res.CustNo.length; m++) {
  var strHtml = '<tr>';
  for (var n=0; n < res.table.length; n++) {
    fieldname = res.table[n][0];
    strHtml = strHtml + '<td>' + res[fieldname][m] + '</td> ';
  }
  strHtml = strHtml + '</tr>'
};
```

Although the processing is far from simple, given we have to transform the column-based structure into a record-based one, the nice thing about this code is that we can adapt it slightly to enable paging, thanks to the built-in filters of the REST server. All you have to do it to pass the "t.r" HTTP parameter, and use the server method executor to increment the page number every time. In this example, you can progress from one page to the next, while in general, you'd want to enable full navigation. Here are the key changes to the previous example, not the complete code:

```
var page = 0;

$('#GetCustomers_paging #button').click (function (){
  var methExec = serverMethods().executor;
  res = methExec.executeMethod("GetCustomers", "GET", [],
    null, true, {"t.r": (page*3+1 + ", 3")}, null);

  res = res.result[0];
  // processing as above

  page++;
  $('#GetCustomers_paging #button').html ("page" + (page+1));
});
```

A situation we can handle quite easily in jQuery is the return of a binary stream for images. All we have to do in this case is to let the server return the binary representation (and not the JSON one) and assign the proper URL to the src attribute of an image tag:

```
$('#GetImage #result').attr ('src',
  '/datasnap/rest/TServerMethods1/GetImage');
```

The final function I want to cover is *GetNamesJson*, which returns a JSON array of strings, built manually on the server (using a *TJSONArray* structure):

```
res = serverMethods().GetNamesJson();
var array = res.result;
var strarray = '';
for (var n=0; n<array.length; n++) {
  strarray = strarray + array[n] + ' ';
};
$('#GetNamesJson #result').val (strarray);
```

Now that we know how to use JQuery to process some of the complex data structures that a Delphi REST server can return, we can choose from the many options available, one that is well suited for generic data entry. In fact, while until now the demo projects have mostly

displayed server side information, I want to go one step further and create an example that can let the client application edit the values of an object. Along with editing, you'll also learn how a Delphi DataSnap REST server can handle update, create, and delete requests by supporting the four key HTTP methods.

## MANAGING OBJECT AND THE HTTP METHODS

Now that we have examined the data types that a Delphi REST server can return and its direct support for *TJSONValue*, and we have at least a basic understanding of jQuery, I want to build an examples that showcases multiple features, including the built-in support for managing of the various HTTP methods supported by a given URL. The idea is not only to retrieve data from the server using the *get* method, but also to let users modify a server side object from a simple Browser-based client written in JavaScript.

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 things 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 implement methods to access it and to change it. Reading the data is what I've done in the MyData function:

```
function TObjectRest.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:

```
{"result": [{  
    "Name": "Sample",  
    "Value": "7016"  
}]}
```

This is the result you get if you point your browser to the URL:

<http://localhost:8080/datasnap/rest/TObjectRest/MyData/Sample>

The URL above makes it quite clear why Delphi's REST implementation requires you to pass parameters in the path. Since a REST URL refers to a resource, and you don't want to write a method for each resource, it makes sense to have the object ID (in this case *Sample*) on the path, but also to pass it as a parameter!

## LISTING OBJECTS WITH TJSONARRAY

Having a list of objects, you might well need to access its elements. For building the user interface, all we need is the names of the objects in the list. 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 TObjectRest.List: TJSONArray;
var
  str: string;
begin
  Result := TJSONArray.Create;
  for str in DataDict.Keys do
  begin
    Result.Add(str);
  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 objects for the user to pick one, we can fully exploit the AJAX model. The HTML page on start up will have no actual data at all, only the HTML elements and the JavaScript code. This is the *empty* HTML:

```
<div>List of objects:
  <br>
  <div id="list" style="padding-left: 20px"></div>
  <br>
  <a href="#" id="refresh">Refresh List</a>
</div>
<hr>
```

As soon as the page is loaded, even without user intervention, the page will make an AJAX call to the server asking for the actual data (in the *refreshList()* function) and populating the user interface. The same function is also installed as an event handler for the *click* event on the *refresh* anchor:

```
$(document).ready(function() {
  // populate the UI
  refreshList();

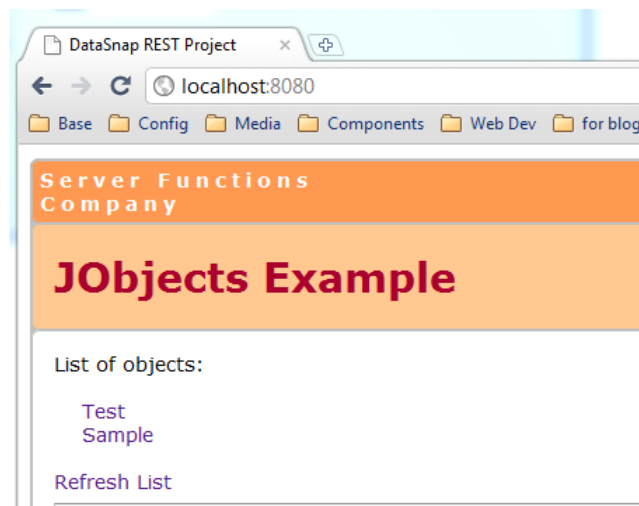
  // redo the operation when refresh is clicked
```

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

The *refreshList()* function 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:

```
function refreshList()
{
    thearray = serverMethods().List().result;
    var theMarkup = [ "<br>"];
    for (var i=0; i < thearray.length; i++) {
        theMarkup = theMarkup + "<a href='#'>" +
            thearray[i] + "</a><br>";
    };
};
```

This is a sample output with the list of the values returned in the JSON array:



There is actually a little more to the code generation. As soon as we have the HTML for the list (which includes many links) we need to connect code to those links so that the client application will load the corresponding server side object in the bottom part of the page for each entry selected. This is accomplished in standard jQuery style by connecting a single event to all anchors (the *<a>* tags) under the *div*. This is the final version of the *refreshList* function with the complete code:

```
function refreshList()
```

```
{
  thearray = serverMethods().List().result;
  var theMarkup = ["<br>"];
  for (var i=0; i < thearray.length; i++) {
    theMarkup = theMarkup + "<a href='#'>" +
      thearray[i] + "</a><br>";
  }
  // add markup to container
  $("#list").html(theMarkup);
  $("#list").find("a").click(function(e) {
    theObj = serverMethods().MyData($("#this").html());
    TMyDataToForm(theObj.result);
  });
};
```

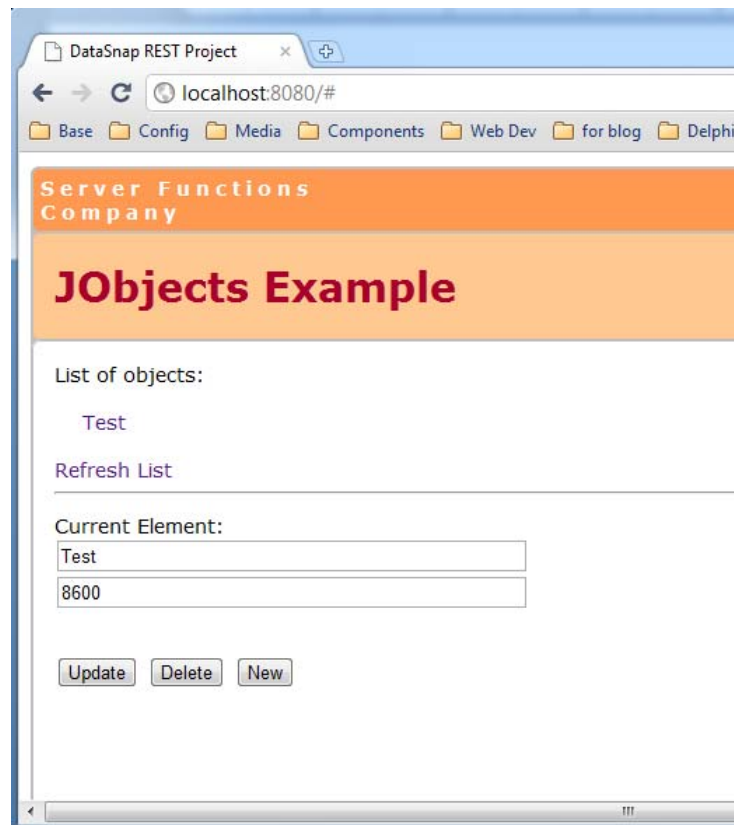
Notice in the code above the use of the `$(this)` expression, which is a reference to the current object or the object that triggered the event (more or less behaving like the `Sender` parameter for a VCL event handler, although in terms of language *this* is more like *self* keyword).

As a user clicks on one the links in the object's list, the JavaScript application will get the object's name from the HTML text and use it as parameter to the call to the `MyData` server function. The user interface for the object data is made of two input boxes, which will later use also for manipulating the object data. The data is moved to these two inputs using the helper function `TMyDataToForm`:

```
function TMyDataToForm (anObj)
{
  $("#TMyData #Name").val(anObj.Name);
  $("#TMyData #Value").val(anObj.Value);
};
```

This code works with the assumption that the user interface has a `div` with the same name of the class and two input elements called like the properties of the JSON object. This is the corresponding snippet of the HTML while the following image shows the complete page in the browser:

```
<div id="TMyData">
  <p>Current Element:
  <br/>
  <input type="text" id="Name" size="50" value="">
  <br/>
  <input type="text" id="Value" size="50" value="">
</div>
```



As you can see above, the program lets us retrieve a value, but has also three buttons to perform the most common operations (the so called CRUD interface – Create, Read, Update, and Delete). This is supported in HTML using the four core HTML methods, respectively PUT, GET, POST, and DELETE. The next section will cover how a Delphi REST server supports these methods.

## GET, 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 general consensus in REST is to avoid using specific URLs for identifying the operations, but to use an URL only to identify server side objects (like MyData/Sample in the URL I listed previously) and to use the HTTP methods to indicate what do to.

Now if Delphi's REST support simply mapped URLs to server class methods, we would have been out of luck (since the same method names would have been used for all four operations). Delphi, instead, maps the URLs plus the HTTP method to a server method. The scheme used is a rather simple: the name of the operation is added before the method name, using this 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 you decide to go with the standard naming rules to support the various operations, you'll have to define your server class like the following:

**type**

```
TObjectsRest = class(TPersistent)
public
    function List: TJSONArray;
    function MyData (name: string): TJSONObject;
    procedure updateMyData (const aName: string;
        jObject: TJSONObject);
    procedure cancelMyData (name: string);
    procedure acceptMyData (const aName: string;
        jObject: TJSONObject);
end;
```

To get or delete an element, we only need the name or ID of the object. To create or update it, we also need to pass the data. In this very simple case, there is only a second value, but in general there could be many. An interesting option (fully supported by DataSnap) is to pass as second parameter an object with all of the fields of the user interface.

The implementation of the three new methods is rather simple and direct 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 (const aName: string;
    jObject: TJSONObject);
var
    md: TMyData;
begin
    md := DataDict[aName];
    md.Value := StrToIntDef (
        jObject.Get('Value').JsonValue.Value, 0);
end;
```

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

procedure TObjectsRest.acceptMyData(const aName: string;
    jObject: TJSONObject);
var
    md: TMyData;
begin
    md := TMyData.Create(aName);
    md.Value := StrToIntDef(
        jObject.Get('Value').JsonValue.Value, 0);
    DataDict.Add(aName, md);
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).

The JavaScript proxy generated by Delphi will have methods names like the server side counterparts (regardless of the fact they use the same URL and a different HTTP method). Another important piece of information is that to pass a JSON object to Delphi all you need to do is to pass a JavaScript object. This is the code of the three buttons:

```
$("#buttonUpdate").click(function(e) {
    var mydata = new Object;
    FormToTMyData(mydata);
    serverMethods().updateMyData(mydata.Name, mydata);
});

$("#buttonDelete").click(function(e) {
    serverMethods().cancelMyData($("#TMyData #Name").val());
    refreshList();
});

$("#buttonNew").click(function(e) {
    var mydata = new Object;
    FormToTMyData(mydata);
    serverMethods().acceptMyData(mydata.Name, mydata);
    refreshList();
});
```

Like for moving data to the user interface, the client program is using a support function to get it back:

```
function FormToTMyData (anObj){  anObj . Name = $( "#TMyData #Name" ). val ( ) ;
anObj . Val ue = $( "#TMyData #Val ue" ). val ( ) ; };
```

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. In this project we are passing back and forth JSON objects, a format easy to process both in Delphi and in JavaScript.

## AUTOMATING THE PROCESS WITH RTTI

Now suppose you have a much more complex data structure with multiple fields. You'll have to create JSON from the Delphi classes and save it back in the server application, and in the browser, map the fields of the JavaScript objects to the HTML user interface elements and vice versa.

In many cases this kind of mapping can be simplified, if not totally automated. For example, you can use Delphi's RTTI (the new extended RTTI available since Delphi 2010) to do this mapping. Given a Delphi object we can create a JSON representation of its public properties with this code (which is part of a *RestPlusUtils* unit I've added to the example):

```
function Obj ectToJson (obj : TObj ect): TJSONObj ect;
var
    context: TRtti Context;
    aProperty: TRtti Property;
begin
    Result := TJSONObj ect. Create;
    for aProperty in context. GetType(obj . Cl assI nfo). GetProperti es do
        Result. AddPai r(TJSONPai r. Create (
            aProperty. Name,
            aProperty. GetVal ue(obj ). ToStri ng));
end;
```

In the real world, you might have to handle special data types (for example dates and Boolean values), but the core idea of the code is clear.

At the opposite, when the server receives a JSON object, it can save this JSON data into a Delphi object (this time taking into account at least some of the data types). Notice that for each property we look for the corresponding JSON value, but if this is missing, no big deal:

```

procedure JsonToObj ect (obj : TObj ect; j Obj ect: TJSONObj ect);
var
  context: TRtti Context;
  aProperty: TRtti Property;
  j Pai r: TJsonPai r;
begin
  for aProperty in context.GetType(obj . Cl assI nfo). GetProperti es do
    begin
      j Pai r := j Obj ect. Get (aProperty. Name);
      if Assigned (j Pai r) then
        begin
          if aProperty. PropertyType. IsOrdinal then
            aProperty. SetVal ue(obj , StrToIntDef (
              j Pai r. JsonVal ue. Val ue, 0)) // not always string
          else
            aProperty. SetVal ue(obj , j Pai r. JsonVal ue. Val ue); // string
          end;
        end;
    end;
end;

```

We can now return and receive Delphi objects mapped to JSON:

```

function TObj ectsRest. MyData(const aName: string): TJSONObj ect;
var
  md: TMyData;
begin
  md := DataDi ct[aName];
  Result := Obj ectToJson (md);
end;

```

```

procedure TObj ectsRest. updateMyData (const aName: string;
  j Obj ect: TJSONObj ect);
var
  md: TMyData;
begin
  md := DataDi ct[aName];
  JsonToObj ect (md, j Obj ect);
end;

```

Written this way, the server can return a JSON version of any object and accept JSON input to update any Delphi object. Furthermore, the program can help you map the JavaScript object (obtained by JSON) to the HTML user interface. Provided you use a *div* with the class name and individual inputs with the property names, you can generate the JavaScript mapping code from within the Delphi application.



The two JavaScript mapping functions shown earlier (*TMyDataToForm* and *FormToTMyData*) are actually generated. The function on the server, called *GenerateMapper*, is based on RTTI and is again part of the *RestPlusUtils* unit. Here is a snippet of its code:

```
sList.Add ('function ' + aClass.className + 'ToForm (anObj)');
sList.Add ('{');
for aProperty in context.GetType(aClass.ClassInfo).GetProperties do
begin
  // outputs JS like: $("#TMyData #Name").val (anObj.Name);
  sList.Add ('    $("#' + aClass.className + ' #' + aProperty.Name +
    '").val (anObj.' + aProperty.Name + ');');
end;
sList.Add ('}');
```

How is this JavaScript code made available to the server? I've edited the *WebFileDispatcher1BeforeDispatch* event handler (which by default takes care of re-generating the *serverfunctions.js* file) and added the following condition:

```
if SameFileName(ExtractFileName(AFileName), 'ui_mapper.js') then
begin
  Handled := True;
  Response.ContentType := 'text/javascript';
  Response.Content := GenerateMapper (TMyData);
end;
```

Rather than updating the JavaScript file before returning it, as the code generated by the wizard does for *serverfunctions.js*, here I'm returning as JavaScript content a dynamically generated string. Now simply add a reference to the URL of this JavaScript resource in the HTML and you'll have the code working and automatically updating itself if you add new properties to your server side object (provided you also update the HTML user interface accordingly).

The previous class I used had only two fields, and manual mapping was quite simple anyway. In the *jObjects* sample project, however, there is also a second class with several properties:

```
type
  TCompanyData = class
  private
    ...
  public
    property CompanyName: string
      read FCompanyName write SetCompanyName;
```

```

property Address: string
    read FAddress write SetAddress;
property City: string
    read FCity write SetCity;
property ZipCode: string
    read FZipCode write SetZipCode;
property State: string
    read FState write SetState;
property Country: string
    read FCountry write SetCountry;
property ContactFullName: string
    read FContactFullName write SetContactFullName;
property ContactEmail: string
    read FContactEmail write SetContactEmail;
property ContactPhone: string
    read FContactPhone write SetContactPhone;
end;

```

Again, there is a server class exposing the four HTTP methods to manage such an object (in this case, there is no list but a single in-memory object, refer to the actual source code for the details of the implementation):

```

{$METHODINFO ON}
type
    TCompanyServer = class (TPersistent)
    public
        function Company: TJSONObject;
        procedure updateCompany (jObject: TJSONObject);
        procedure cancelCompany;
        procedure acceptCompany (jObject: TJSONObject);
    end;

```

I've added to the web module a new DataSnap server class connected with *TCompanyServer*, a new action connected with a new page producer, and the class support in the *uimapper.js* generated code.

Now all I have to do to create the user interface is to clone the previous HTML file and replace the data entry form with one having the proper input fields. In the new HTML template, *company.html*, I've created a bare-bone data entry form with buttons performing the four operations above. In case you are really lazy (or as starting point of the development), the server could also generate a bare-bone form, generating a custom user interface. This is a technique used by other web development environments and generally called *scaffolding*.

As a very preliminary demo, I've added to the *RestPlusUtils* unit a *HtmlFormForClass* function, which is activated using a special HTML tag:

```
<!-- automatic form -->  
<#scaffolding class="CompanyData.TCompanyData">
```

This tag is processed on the server, extracting the attribute value which holds the qualified name of the class (unit name and class name) as used by the new RTTI to find a class with no registration requirements:

```
// in TWebModule2.ServerFunctionInvokerHTMLTag  
else if SameText(TagString, 'scaffolding') then  
begin  
  ReplaceText := HtmlFormForClass (TagParams.Values['class']);  
end;
```

While in the HTML file there is also a custom version with slightly better formatting, here is a screenshot of the form generated using the *scaffolding* technique:

Server Functions  
Main Page

## JObjects Example: Company Pa

CompanyName	<input type="text"/>
Address	<input type="text"/>
City	<input type="text"/>
ZipCode	<input type="text"/>
State	<input type="text"/>
Country	<input type="text"/>
ContactFullName	<input type="text"/>
ContactEmail	<input type="text"/>
ContactPhone	<input type="text"/>

Demo by Marco Cantù

(Before you ask, yes, I'm working on an extended version of this core architecture, which will include specific support for more data types, extensive browser-based validation, and other nice goodies: Feel free to get in touch for more information).

## 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 the REST support in DataSnap is primarily focused on remote method calls. You can return a dataset from a REST server, but the column-based format combined with the fact there is no specific support for the user interface, makes the dataset management support somewhat limited. I did show you how to create a simple HTML table on top of this format and also hook to the paging support.

As we have seen, however, it is quite simple to return a custom JSON data structure from Delphi's REST architecture, so besides improving the user interface we might be able to make things simpler by customizing the server side processing. Also, rather than building a custom HTML user interface we might want to take advantage of existing data grid components. That's the focus of this last section and the related example.

The server program is quite bare-bone, as all it does is to return the data of a complete Dataset, with or without metadata and in different JSON formats. For the creation of a custom HTML table based on the dataset, I've created two separate functions returning the list of fields and the entire dataset in a JSON array, with an object for each record:

```
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;  
  
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;

```

To reduce the amount of data I could have avoided repeating the field name, returning an array for each record, or skipped the null or empty fields. These two methods are invoked by the client application after loading the *jRestClient.html* page, building an HTML table dynamically. In the HTML file, in fact, there is only the table structure:

```

<table id="result" border="1" cell spacing="0">
  <thead id="head"></thead>
  <tbody id="body"></tbody>
</table>

```

The client application first loads the metadata and processes the array creating the table header row:

```

var theMetaArray = serverMethods().Meta().result;
var headMarkup = "<tr>"; // table header
for (var j=0; j < theMetaArray.length; j++) {
  headMarkup = headMarkup + "<th>" + theMetaArray[j] + "</th>";
};
headMarkup = headMarkup + "</tr>";
$("#head").html(headMarkup);

```

The second step is to get the actual data and use the metadata array to navigate it. In this case, the JavaScript code accesses the object properties dynamically, with the notation *object["propertyname"]*. This means that rather than using the property symbol (as in *thearray[i].Company*) the code uses the property's name read from the metadata array (*theMetaArray[j]*), resulting in the expression:

```

thearray[i][theMetaArray[j]]

```

This is the complete JavaScript code used to create the table's body markup:

```

thearray = serverMethods().Data().result;

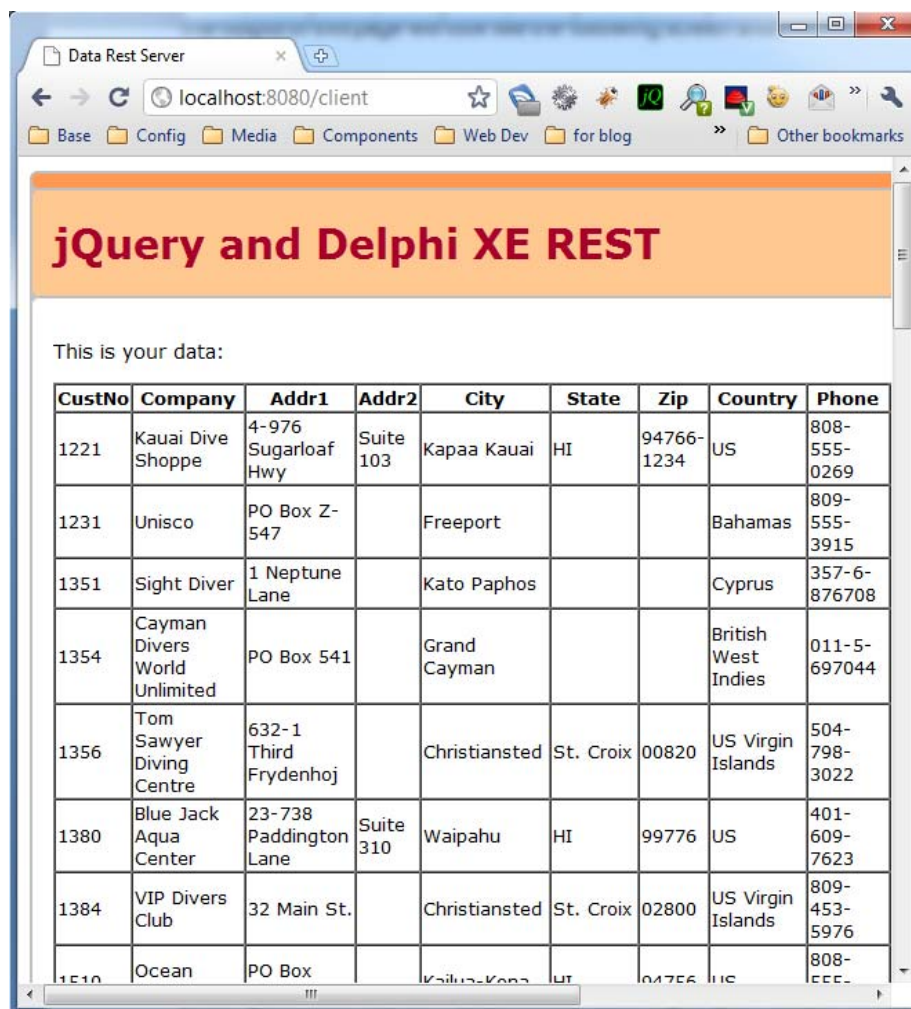
```

```

var bodyMarkup = ''; // table content
for (var i=0; i < thearray.length; i++) {
    bodyMarkup = bodyMarkup + "<tr>";
    for (var j=0; j < theMetaArray.length; j++) {
        bodyMarkup = bodyMarkup + "<td>" +
            thearray[i][theMetaArray[j]] + "</td>";
    };
    bodyMarkup = bodyMarkup + "</tr>";
};
$("#body").html (bodyMarkup);

```

The output of this page will look like the following screen shot:



**jQuery and Delphi XE REST**

This is your data:

CustNo	Company	Addr1	Addr2	City	State	Zip	Country	Phone
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI	94766-1234	US	808-555-0269
1231	Unisco	PO Box Z-547		Freeport			Bahamas	809-555-3915
1351	Sight Diver	1 Neptune Lane		Kato Paphos			Cyprus	357-6-876708
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cayman			British West Indies	011-5-697044
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj		Christiansted	St. Croix	00820	US Virgin Islands	504-798-3022
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310	Waipahu	HI	99776	US	401-609-7623
1384	VIP Divers Club	32 Main St.		Christiansted	St. Croix	02800	US Virgin Islands	809-453-5976
1510	Ocean	PO Box		Kailua-Kona	HI	96756	US	808-555-

## USING THE DATATABLES COMPONENT

Again, this serves only as a starting point, since rather than creating an HTML table manually and adding some basic behavior to it, I can take advantage of one of the jQuery plug-ins that would add significant capabilities to the HTML table, turning it into a powerful grid, with sorting, filtering, and other capabilities.

One such component is DataTables, a jQuery plug-in available and documented at [www.datatables.net](http://www.datatables.net). (I picked this specific jQuery plug-in because I have some experience with it, but there are other equally powerful grid components for jQuery, among the many plug-in extensions for this library).

There are many ways to pass the data to the DataTables component, one of the simplest is to initialize it with a JSON structure, including in a single object both data (*aaData*, an array of arrays) and metadata (*aoColumns*, an array of objects describing the column titles and formatting style). Therefore, I've added to the server side class a specific method returning the specific JSON format:

```
function TServerData.DataTables: TJSONObject;
var
  dataArray, subArray: TJSONArray;
  metaArray: TJSONArray;
  I: Integer;
begin
  ClientDataSet1.Open;
  dataArray := TJSONArray.Create;

  while not ClientDataSet1.EOF do
    begin
      subArray := TJSONArray.Create;
      for I := 0 to ClientDataSet1.FieldCount - 1 do
        subArray.Add(ClientDataSet1.Fields[I].AsString);
      dataArray.AddElement(subArray);
      ClientDataSet1.Next;
    end;

  metaArray := TJSONArray.Create;
  for I := 0 to ClientDataSet1.FieldDefs.Count - 1 do
    begin
      metaArray.Add (TJSONObject.Create(
        TJSONPair.Create('sTi tle', ClientDataSet1.FieldDefs[I].Name)));
    end;

  Result := TJSONObject.Create;
  Result.AddPair(TJSONPair.Create('aaData', dataArray));
```

```
Result.AddPair(TJSONPair.Create('aoColumns', metaArray));
end;
```

With the component (the inclusion of its JavaScript source file) and this JSON structure available, the client side code becomes extremely simple. You need a form with a table (for simplicity I've created one with a fixed set of headers) and these two lines of initialization code to apply the *dataTable* method to the HTML table while passing to it the JSON data:

```
var theDataTablesData = serverMethods().DataTables().result;
$('#example').dataTable(theDataTablesData);
```

The result will look like the following image. In this data grid, you can select the number of visible rows (in this case the paging is done on the client, but you can configure the component for server side paging), sort the content on any column by clicking on its title, and filter the visible rows using a search box (the filter is applied to all columns by default):

#### DataTables via Ajax

Show  entries
Search:

CustNo	Company	Addr1	Addr2	City	State	Zip	Country	Phone	FAX	TaxRate	Contact	LastInvoiceDate
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI	94766-1234	US	808-555-0269	808-555-0278	8.5	Erica Norman	2/2/1995 1:05:03 AM
1231	Unisco	PO Box Z-547		Freeport			Bahamas	809-555-3915	809-555-4958	0	George Weathers	11/17/1994 2:10:33 PM
1351	Sight Diver	1 Neptune Lane		Kato Paphos			Cyprus	357-6-876708	357-6-870943	0	Phyllis Spooner	10/18/1994 7:20:30 PM
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cayman			British West Indies	011-5-697044	011-5-697064	0	Joe Bailey	1/30/1992 2:00:56 AM

The DataTables component has several other capabilities, including even editing support through another jQuery plug-in,

## 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 XE. 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 demo 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 *Handbooks* on the latest versions of Delphi (Delphi 2007 Handbook, Delphi 2009 Handbook, and Delphi 2010 Handbook).

Marco is a frequent conference speaker and teaches advanced Delphi classes (including Web development with Delphi) at companies worldwide. He also organizes public training classes in Italy through his company (Wintech Italia Srl) and has partnered with Cary Jensen to organize the Delphi Developer Days events in the US and Europe ([www.delphideveloperdays.com](http://www.delphideveloperdays.com)). Marco offers also consulting, mentoring, code review, and architectural review of Delphi applications.

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](mailto:marco.cantu@gmail.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](http://www.embarcadero.com).