

# Development and Deployment of Delphi Multi-tier Applications

Marco Cantù, Delphi Product Manager

December 2012

---

**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

## INTRODUCTION

This paper guides a Delphi developer through the DataSnap technology in Delphi XE3. The focus of this document is to offer an overview of the technology and its usage and deployment scenarios, more than a detailed technical analysis of the available features of this multi-tier library. I'll cover the options you have in terms of integration with web servers and hosting in the cloud. I'll discuss how to make these services more scalable and robust, and offer a very simple overview of the different types of clients you can build with Delphi and other tools.

More specifically, the paper is divided into 4 sections:

- Building application services with DataSnap
- Deploying services, integrating with web services, and hosting them in the cloud
- Making services scalable and robust
- Accessing application services from different types of clients

At the end I'll be able to provide some assessment of DataSnap usage scenarios, providing practical tips and suggestions.

Notice that if you are interested in a step by step introduction of the DataSnap technology you can refer, among other sources, to Bob Swart's material listed on <http://www.embarcadero.com/rad-in-action/datasnap>, while for an introduction of REST in Delphi you can refer to my presentation and paper at <http://www.embarcadero.com/rad-in-action/datasnap-rest> (but I'll repeat some relevant material of that older paper in this document).

## PART I: BUILDING APPLICATION SERVICES WITH DATASNAP

In the first part of this paper, I'll introduce the development of DataSnap servers, trying to guide you through the different types of servers available and their respective features and

goals. As you'll soon find out, in fact, DataSnap is more a collection of technologies than a single one, and this causes some confusion to newcomers.

For a long time Delphi has included a technology for building multi-tier database applications. Formerly known as MIDAS, the framework was later renamed DataSnap. Delphi's multi-tier technology was originally based on COM, with the remoting capability offered by DCOM, but with the ability of bridging the COM servers and expose them through other connectivity layers like CORBA, TCP/IP, and later also SOAP.

The DataSnap framework was completely rewritten in Delphi 2009. This newer architecture is still in use today, is more flexible, and has the ability to plug in connectivity layers. All of the original dependencies on COM have been removed.

A DataSnap server is built around different components, DSServer, DSServerClass, and one of the available connection components. Let me start with the connection alternatives, as this offers a broad scenario of the capabilities of DataSnap.

## THE CONNECTIVITY OPTIONS

A DataSnap service can be based on a large number of server applications. Here for the moment I want only to focus on the options you have in terms of connectivity, delaying to a later section of the paper the deployment alternatives. In terms of connectivity, a DataSnap application can use:

- A TCP/IP connection component (DSTCPServerTransport), offering a state-full and stable connection to the clients. This is generally suited for an internal network, but given the recent addition of connection monitoring capabilities could also be deployed to the Internet as a whole. The socket-based connections offer custom data compression and data encryption, through filters that must be deploying on both the server and the client. This model makes sense only if the client application is also written in Delphi or C++Builder.
- An HTTP connection component (DSHTTPService), offering a state-full connection on top of a stateless protocol, though custom session management and the ability of keep server side object active in the session (as described later). Again, this is mostly meant to be used when the client is also a Delphi or C++Builder application, and can use the HTTPS protocol for extra security.
- A REST connection, based either on the same HTTP connection component described above or on the integration of the DataSnap server with the WebBroker architecture (which offers deployment as an IIS module or stand-alone web servers). In both cases the DataSnap server behaves as a stateless server, and there is much more flexibility in terms of the development tools you can use to build the client applications. This is

likely the most open, flexible, and scalable option and the one I'll keep my focus for most of the paper.

Before we look at the details of how to build these applications and which features they provide, I have to cover the various DataSnap server side components and introduce sessions. In any case, keep in mind that the three models above could all be surfaced by a single server application, given that a single DataSnap server can offer multiple connection types (sockets and HTTP) and a single HTTP connection has dual interfaces (HTTP and REST)

## THE DSSERVER COMPONENT

DSServer is the main server configuration component, which is needed to wire all the other DataSnap components together. This component handles connections and manages callbacks, which is why you have to make sure there is only one server component if you need a unified callback mechanism.

Notice, however, that in case of web applications you need to create a specific data module hosting a DSServer component (something you can achieve by picking one of the wizard's options) or else you have one server for each web module, which in turn is created when there is a new HTTP request.

## THE DSSERVERCLASS COMPONENT AND THE LIFETIME OF SERVER OBJECTS

DSServerClass is a component needed for each class you want to expose to client applications. Each services exposes methods and data using one or more server classes. The DSServerClass component is not the class you make available, but acts as a class factory to create objects of the class you want to call from a remote client. In other words, the DSServerClass component will refer to the class that has the public interface.

This component has a second key feature, the indication of the lifetime of the objects of the target class. There are three alternatives for the lifetime:

- Invocation lifetime indicates that for each invocation the service will create a target object, call the specific method, and than immediately free the object. In other words, the server will create an object for each request (regardless of the user and the session), and avoid caching them, in a model best suited in case of a stateless architecture based on the HTTP and REST protocols.
- Session lifetime indicates that the object is creates at the first invocation and kept in memory while the client has an active session. A following request from the same client and the same session will use the same server side object. I'll describe in the following paragraph how DataSnap manages sessions, clarifying this model.

- Server lifetime indicates that the object is a global object created by the server the first time there is a request for it, used for any request to that object by any other client, independently from the session. This object is kept in memory indefinitely.

The lifetime model has a direct effect on the concurrency model. For an object with server lifetime, multiple threads activated by different connections at the same time might use the same server object, and you are responsible to provide safe multi-threaded access to the server object. In general, you should stay away from server lifetime, particularly if your goal is scalability.

For invocation lifetime, this issue does not exist, given each thread will have its own server object. For session-based objects, the problem arises only if the client can make multiple requests using multiple threads at the same time (which is not a common practice, but might happen).

Now, what is relevant to consider is that not all lifetime models are available for each type of connection. In particular, the session lifetime for server side objects is not available for REST servers. The reason is that server side objects could have a large memory footprint and should not be kept in memory for a long session that you have to keep around long after the client has done the last request.

## SESSION AND OBJECT LIFETIME

The invocation model is strictly tied to session management. By default DataSnap creates in-memory session objects for each user initiating a connection to the server:

- In case of a socket-based connection, the session is tied to the connection itself. The user connects, invokes methods causing the creation of server side objects, further method calls will hit the same objects, and as the user disconnects his server objects are deleted.
- In case of stateless HTTP-based connections, the session is based on an ID returned and passed back again as an extra HTTP header. In this case, the session remains active for a given amount of time (20 minutes by default) after the last request from the given client. This can cause some memory overhead, given objects remain in memory long after the user has stopped invoking server methods. As an alternative to keep track of the user and application status in server side objects, you can use the session object itself.
- In case of a pure REST model, the mechanism is similar to the HTTP scenario but the client is responsible for handling the connection (using this extra header, a cookie, or a query parameter) or else the system will create a new session for each incoming

request, wasting memory and resources. In this scenario, the session object is the only storage tied to the user session, given that server side session lifetime is not available.

These session objects (based on different server classes depending on the connection model), store key information like the login status, the username used for login, the roles used for methods authentication, and so on. As a developer, you can add more information to the current session object, which can be accessed directly in any server method. Again, this is the standard way to keep track of user and application information in case of REST connections, but it is also the suggested way for HTTP-based applications that you might want to scale.

## DATA SNAP DUAL INTERFACE MODEL

Before we (finally) start looking at demonstrations, there is one more key factor to consider. Historically MIDAS and DataSnap were focused on exposing database tables to client applications. This is done using a specific interface, called `IAppServer`, which is exposed by “remote” data modules. You do not implement these methods directly, but expose one or more `DataSetProvider` components, which are used by the remote data module to respond to the methods of this `IAppServer` interface. On the client side, you can use specific connection components (`DSProviderConnection`) and the `ClientDataSet` component to connect to the `IAppServer` interface and the tables exposed by the providers.

On top of this, DataSnap offers the ability to expose custom methods. Originally done via COM, the remote method invocation now uses a custom layer, based on server classes, RTTI, and some specific custom rules for parameter passing. The server methods can use core data types (integers, strings, characters, floating point numbers, and so on) but also complex ones like datasets, streams (sequences of binary data), JSON data structures, and even entire Delphi objects through some marshaling classes.

Now, what is important to realize is that not all DataSnap servers support this dual model. The `IAppServer` interface, in fact, is strictly tied to a state-full model, as it has concepts like “next data packet”. This is why the `IAppServer` interface and the `DataSetProvider` models are not available for stateless REST applications build with DataSnap.

Given that a REST service can expose tables, though, you can still move database data easily from a server to a client, but if you have experience with the `ClientDataSet` – `DataSetProvider` combination you probably know how fast and easy they can make the development of multi-tier applications with master-detail and other complex structures. Implementing the same approach in a DataSnap server currently requires more effort, which is a good reason to keep the older model around and support it.

So, on one side we have a traditional, more RAD, simpler, Delphi development model, on the other side we have the more open, flexible, and scalable REST approach. That is why, before

we finally proceed with the actual development, there is one more element to cover in this long introduction, the REST development model.

## 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. Another web service solution is the use of a REST (Representational State Transfer) approach. The introduction of this formal name and the theory behind it are fairly recent. What is relevant to mention up front is that there is not 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 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.

Having said that there is not a REST standard and that you need no specific tools for REST development, there are 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.



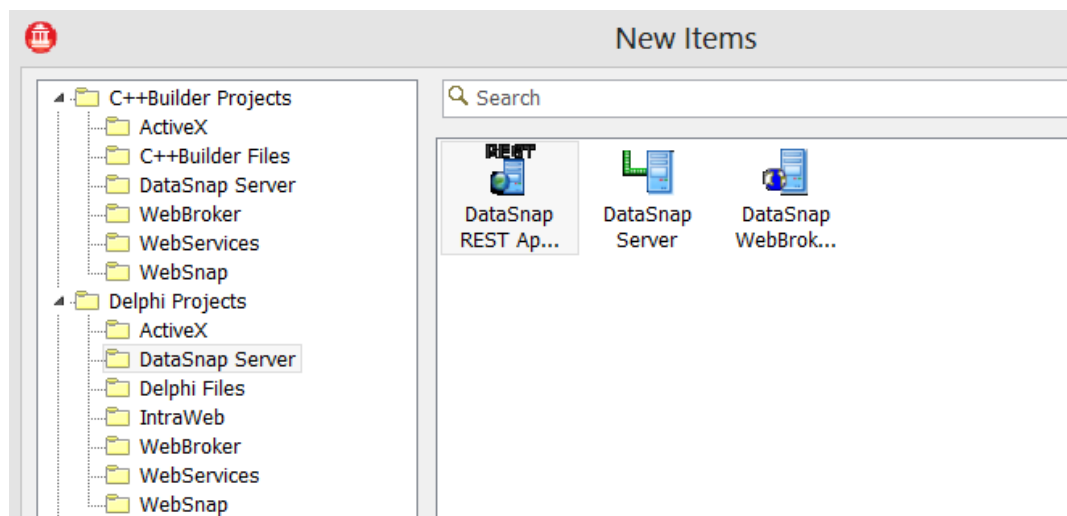
The HyperText Transfer Protocol is the standard at the heart of the World Wide Web, and needs no introduction. Granted, HTTP can be used not only by Web Browsers, but also by any other application. In Delphi applications the simplest way to write a client application that uses HTTP is to rely on the 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. The component supports the various HTTP methods, beside Get.

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 even Apache modules). The WebBroker architecture supports both models (as we'll see in more detail).

Moving to DataSnap, as I already introduced, you can create DataSnap REST servers using the internal HTTP connection component (based on the Indy HTTP server) or using WebBroker (which supports ISAPI DLLs or stand-alone servers still based on the Indy HTTP server). You cannot use CGI as this will not provide sessions management, and you can manually wire the available units to support Apache modules. Notice, though, that I tend to deploy my servers on Apache using a different model I will cover in the section on deployment.

## BUILDING A DATASNAP SERVER

After this long and detailed introduction providing the key scenario, we can start building some DataSnap services. If you select the File | New | Other menu item in the Delphi IDE, and select the DataSnap server node, you'll see three different options:

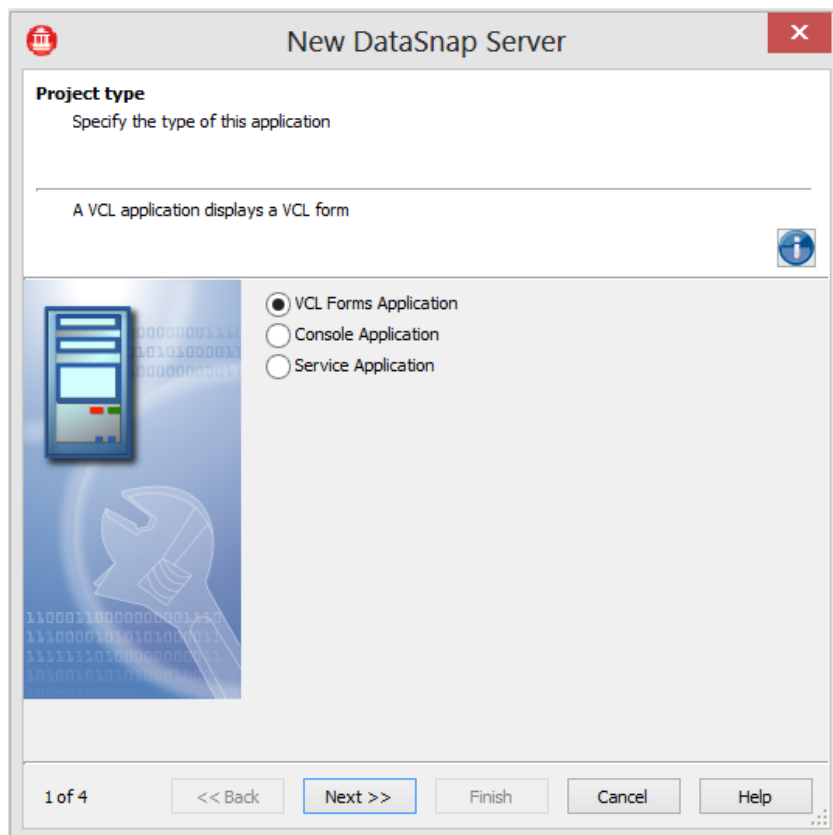


Going from the most traditional DataSnap model to the more open one, they are:

- **DataSnap Server**, which supports both TCP/IP and HTTP based connections to a more traditional DataSnap server
- **DataSnap WebBroker Application**, which has an architecture tied to the WebBroker model allowing integration with IIS via ISAPI, with Apache, or also the deployment of a stand-alone web server.
- **DataSnap REST Application**, which is also tied to WebBroker and adds support for the basic structure of a Web application (including basic HTML and JavaScript files).

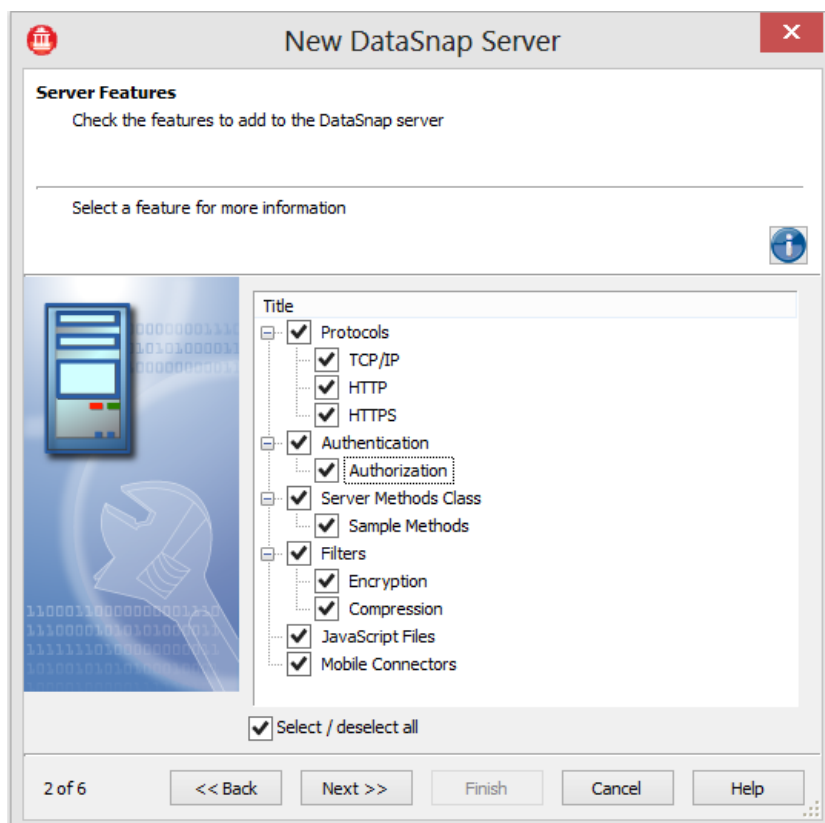
As you can see from the long introduction above, different types of DataSnap servers have different features, but the three wizards also allow for multiple options, so they do not have a direct one-to-one match with the available architectures.

In parallel, a single model can be hosted by different type of applications (console applications, standard Windows applications, or service). Let's start from the "classic" DataSnap server. Here the first page lets you pick the Windows applications structure:

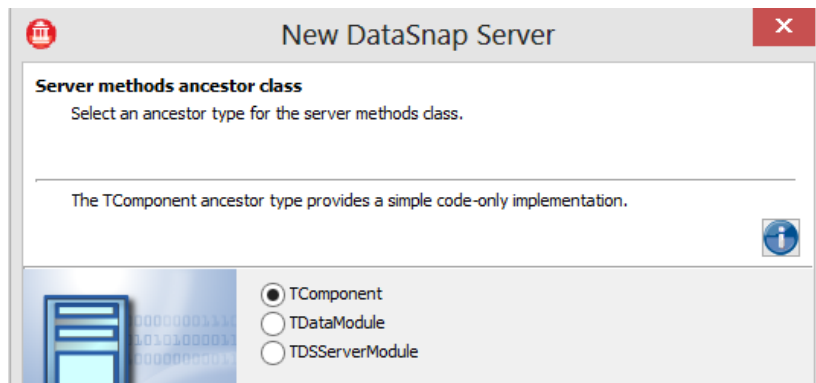


While this has an impact on the deployment scenario, it has little effect on the internal structure and capabilities of the DataSnap server. In general, I tend to use a VCL application for testing and debugging, and might later move it to a Service application for more stable deployment. Given you can change the structure by picking a different project file, you can change the project structure from one to the other of these three models later on.

The second page is where the DataSnap options start to surface, with the ability to pick the server connectivity (TCP/IP, HTTP, and/or HTTPS) and several other features of the DataSnap server application. Most of these features (like compression or authentication) you can easily add later using specific components, while others (like support files) are added by the Wizard to the projects and requires a little more manual work to configure later. Here I have picked all options, but in my actual demo I have omitted HTTPS.



The following page lets you customize the port of each of the selected connections, while the following (optional) page is for HTTPS certificates. The second last page is for picking the target class is the first ServerClass component (you can freely add others later in the code):

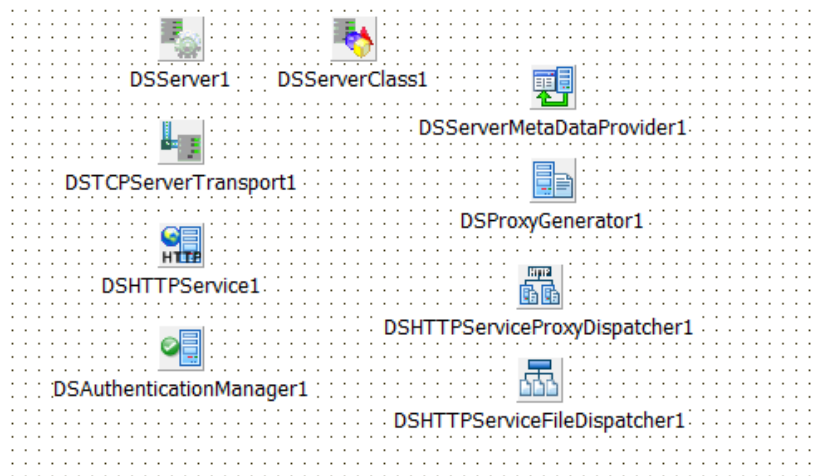


Here are the options are a plain TComponent base class, a TDataModule (a container of non-visual components like database tables and database connections), and a TDSServerModule base class, the special class implementing the IAppServer interface and exposing directly any DataSetProvider component added to itself.

The last option is the only one you can use for connecting a ClientDataSet directly to your server, the first one is where you write all of the code, while the data module option is an in-between, offering visual components hosting but no hardwired connection or ready-to-use method. As mentioned earlier and covered again later on, for a stateless REST application you cannot use the DSServerModule.

Finally, the last page of the wizard lets you select an existing or new folder where the wizard will place all the source code files and support files of the generated application. Notice only that the folder name will also be used as project name, so it must be a valid Pascal language identifier (i.e. it cannot start with a number or have spaces).

After generating the application (again, with all feature active but HTTPS), you will get a project with a totally useless main form, a global data module with the DataSnap application structure, and a second data module (as a plain unit) used as target server class. The global data module with the configuration components will look like this:



Many of the settings of these components store the application configuration. For example, the default ports you select for connecting to the server via TCP/IP and HTTP become properties of the respective components. I will get back to some of these components and their settings later on and again there are already some existing DataSnap tutorials that cover these. Moreover this application exposes mobile connectors using the Proxy generator and dispatcher and supports client JavaScript invocation using the Proxy generator and the HTTP service file dispatcher.

This might be a bit confusing. If we have built a “classic” server using this wizard, how come we have support for features of a DataSnap REST application? It turns out the “classic” wizards let you build all types of DataSnap server, with the exclusion of the WebBroker integration model. By unselecting (or later removing) TCP/IP support we would have a perfect HTTP REST server. In other words, when you want to build any type of DataSnap server not tied to WebBroker but based on the internal connection components, this is the model you should follow.

## SHORT OVERVIEW OF THE WEBBROKER ARCHITECTURE

Before we look into the development of a WebBroker DataSnap server, it is important you have an idea of what I’m talking about. This section offers a short introduction of a technology that has existed since Delphi 3, for those who never used it or used it only occasionally. If you already used WebBroker you can certainly skip this section.

The WebBroker technology, available in Delphi since the early days of the product, is a framework to let you create Web server extensions that can be deployed as applications, libraries, and (even if unofficially) modules. There is a fourth option, which is the use of a debug tool, called Web App Debugger as a replacement for a web server while developing and debugging the application.

A WebBroker application is built around a designer, which has an object holding the web request received from the client and the web response, plus a collection of actions tied to the incoming URLs. This derives from `TCustomWebDispatcher`, which provides support for all the input and output of your programs and defines the events and properties.

These properties are defined using a base abstract class, but an application initializes them using a specific object (such as the `TISAPIRequest` and `TISAPIResponse` subclasses for an ISAPI library). These classes make available all the information passed to the server, so you have a single approach to accessing all the information. The key advantage of this approach is that the code written with WebBroker is independent of the type of application (CGI, ISAPI, Apache module); you will be able to move from one to the other, modifying the project file or switching to another one, but you will not need to modify the code written in a WebModule.

To write the application code, you can use the Actions editor in the WebModule to define a series of actions (stored in the `array` property) depending on the *path name* of the request. This path name is the portion of request URL that comes after the program name and before the parameters.

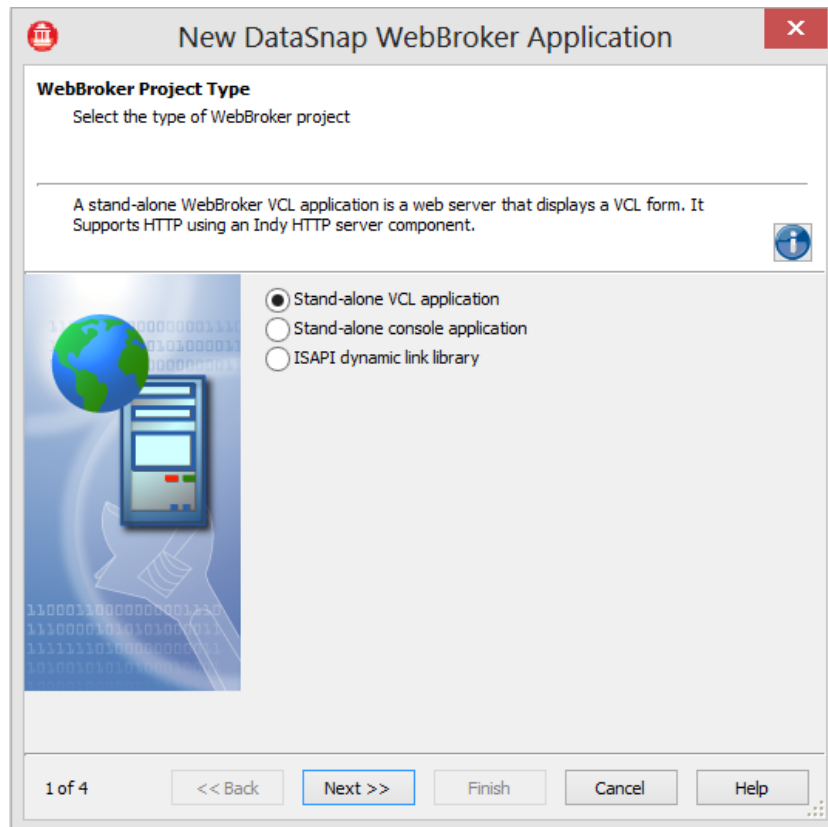
By providing different actions, your application can easily respond to requests with different path names, and you can assign a different producer component or call a different event handler for each and every possible path name. In the event handler, you write the code to specify the response to a given request, in the simplest case returning some HTML in a string:

```
procedure TWebModule1.WebModule1WebActionItem1Action (  
    Sender: TObject; Request: TWebRequest;  
    Response: TWebResponse; var Handled: Boolean);  
begin  
    Response.Content :=  
        '<html><head><title>Hello Page</title></head><body>' +  
        '<h1>Hello</h1>' +  
        '<hr><p><i>Page generated by Marco</i></p>' +  
        '</body></html>';  
end;
```

Needless to say there is much more in the WebBroker architecture, but that would go beyond the scope of this paper, focused on using WebBroker as a foundation of the DataSnap framework.

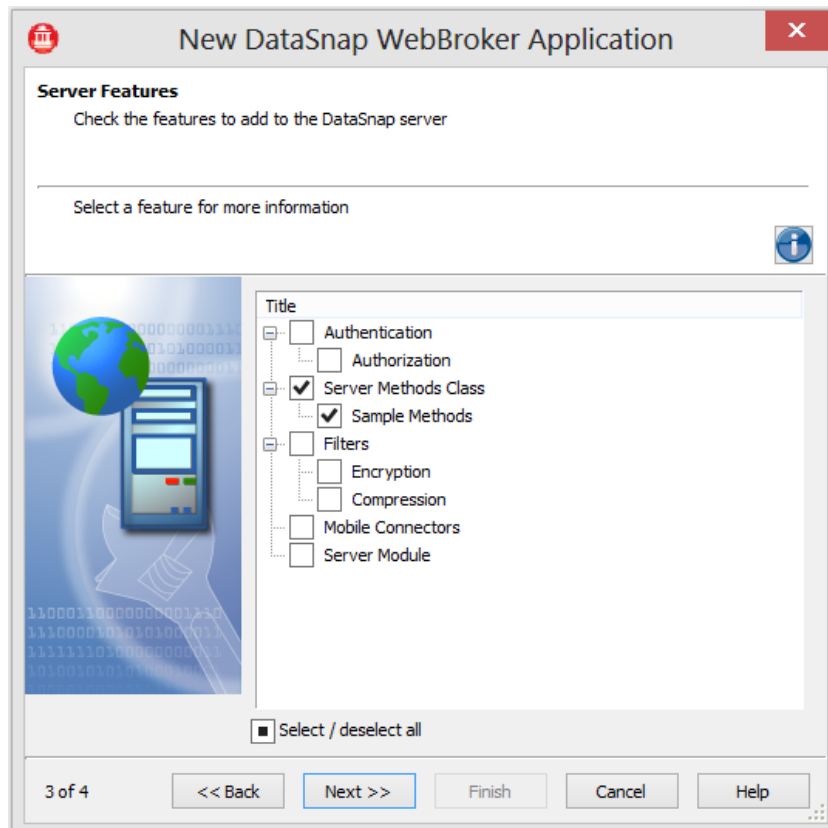
## BUILDING A WEB BROKER SERVER

In fact, if we use the second wizard the DataSnap REST Application, or the third one, the DataSnap REST Application, we obtain in both cases a WebBroker-based DataSnap server. The second wizard has the following initial options, different from the first page described earlier:



Here the options include two different types of stand-alone applications based on the IdHTTPServer component (with a main form or a console structure) or the ability of creating an ISAPI DLL for integration with Microsoft's IIS server. Although not officially supported, you can adapt this DLL (including different source code units) and build an Apache module. Again, switching between these three options can be done later by generating a different project file and copying one or two units to your existing project and then merging them.

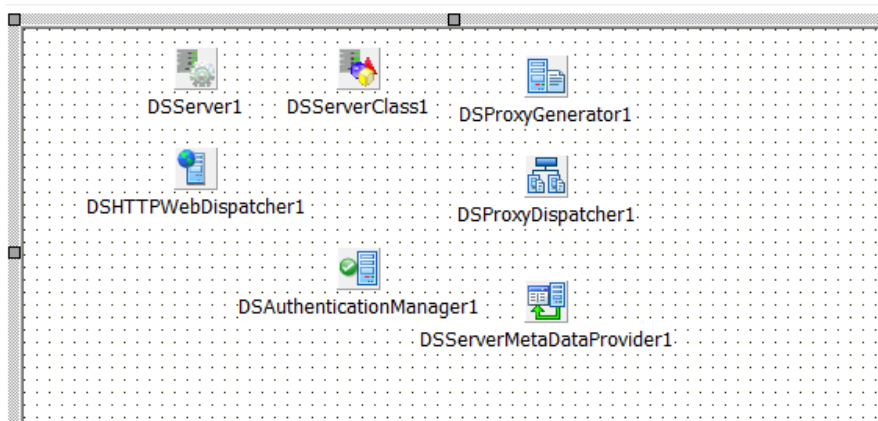
If you pick one of the two stand-alone options, the following information will let you configure the port, as usual, while the third page has the actual DataSnap features selection:



As you can see, this is a subset of the features available in the classic DataSnap server application: The only missing elements are the protocols, as in this case we are integrating with an HTTP server and that is the only option. For HTTPS support, you would enable that feature at the Web server configuration level, if available. Another missing feature is the deployment of JavaScript files, since this is what the third wizard, DataSnap REST Server, is meant for.

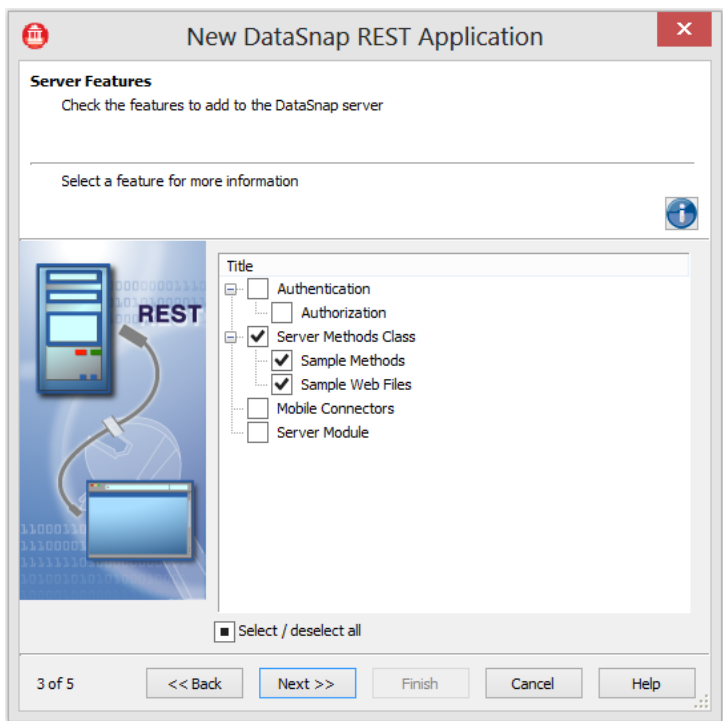
There is also an addition to the previous set of options, and that is the ability to generate a separate server module hosting the DSServer component and the DSServerClass: As I already mentioned, this option lets you have a global module with these components rather than creating a new copy for each Web module generated by the web server to respond to any web request. If you don't select this option, you end up with a web module similar to the global data module of the previous application:





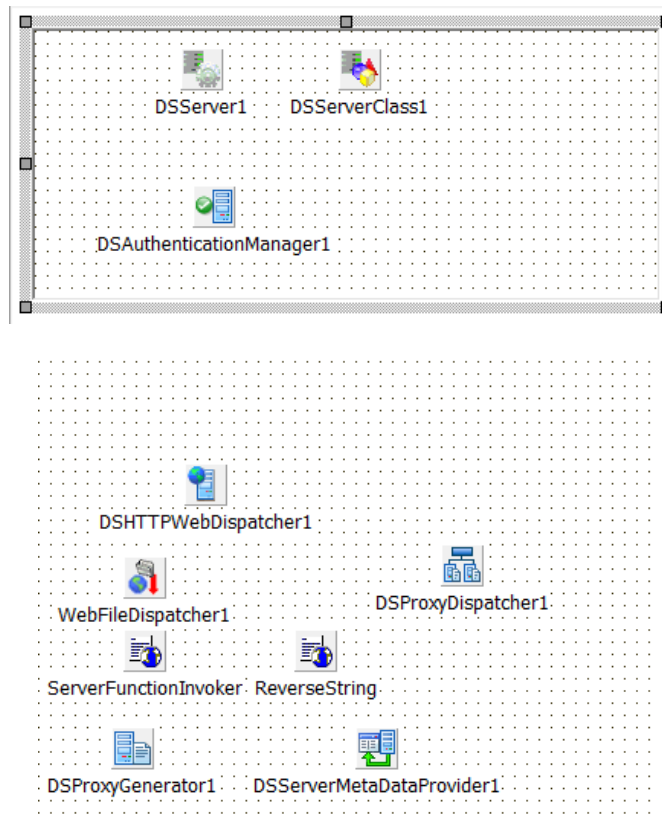
Here the HTTP connection component (DSHTTPService) is replaced by a Web Dispatcher component, which filters some of the incoming requests by URL (basically anything starting with “/datasnap”), and passes them to the DataSnap framework for processing. Given that this is a WebBroker application, the web module has also a list of actions tied to other web document paths.

If we choose the third wizard, the DataSnap REST Server, the first page is identical to previous one (with the stand-alone servers and the ISAPI DLL), while the DataSnap features selection changes:



In this case the filters are not available (as they require a custom Delphi or C++Builder client application), while there is a new specific option related with the sample web files to be added to the project, basically providing a small web browser application (based on HTML and JavaScript) along with the REST server. Again, we have the option to generate a server module, which I recommend using.

As with the server module, the wizard generates two modules, this data module and a separate web module which hosts some of the components each:



The web module has a few extra components used for the Web application, like the two page producer components (connected with HTML files) and the `WebFileDispatcher`.

By now you may be a bit confused with all of the options and the different ways to build DataSnap server applications. Again, there are two overall models, stand-alone or WebBroker integrated. Stand-alone can use sockets or HTTP. WebBroker integrated can be a server extension DLL or a stand-alone web server. And the various DataSnap features can or cannot be used depending on the architecture you have picked. While this is certainly a bit confusing,

the three wizards make quite a good job in helping you generate the proper structure for the different types of projects.

You might ask yourself which model you should use. This depends on many factors, like the type of client applications (Delphi or not only), the scalability (sockets vs. HTTP), the overall traffic, the stability of the connections and of the client applications... and much more. As we start focusing on deployment options and client applications development, things should become a little clearer.

## PART II: DEPLOYING SERVICES

So we have now built a few different DataSnap server applications, using the wizards. What is next? We should now compile and test them, see if they work, and examine the various deployment options that we have available. In this second and shorter part of the paper I will focus on deploying these DataSnap servers, on integrating them with Web Services, and on hosting them in the cloud (a topic with very limited technical differences, but a significant role).

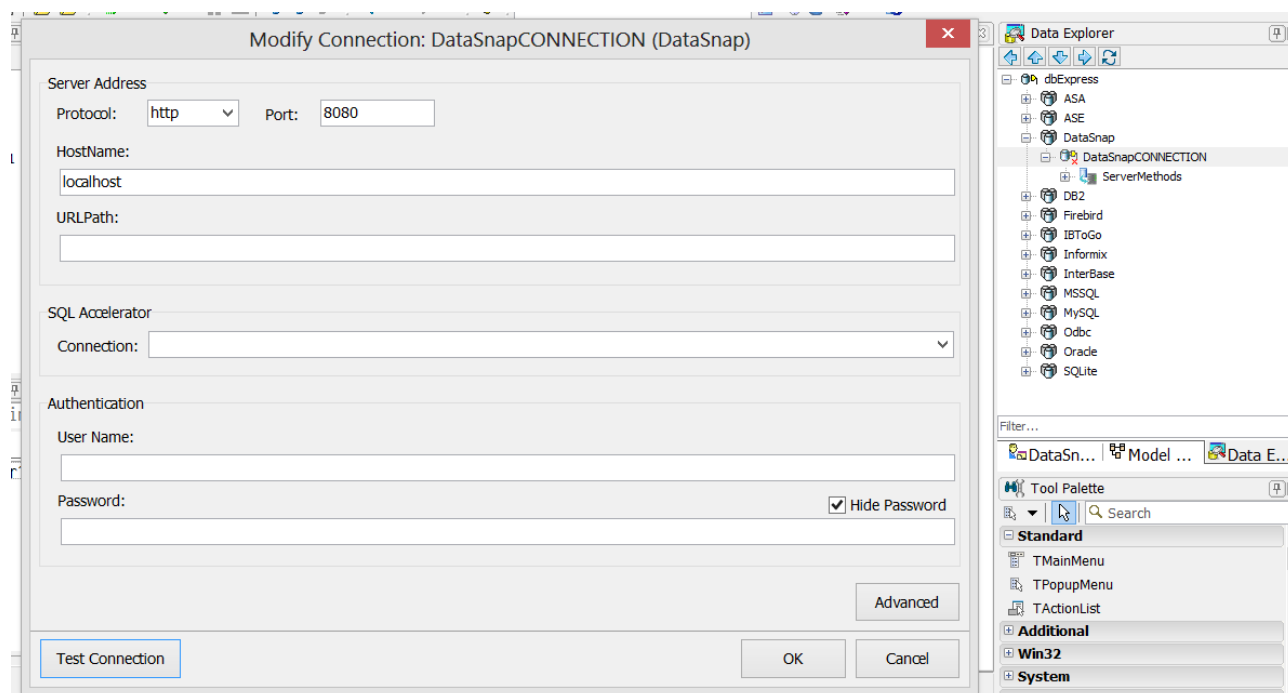
Before we delve into deployment, let me start me with a basic step: compiling the service applications in Delphi and making sure they work.

### COMPILING AND TESTING DATASNAP SERVERS

We can now compile each of the three servers and check that they work. Given they have different interfaces, these operation will also be different. The classic DataSnap server (the project generated by the first wizard) has three different interfaces, sockets, HTTP, and REST.

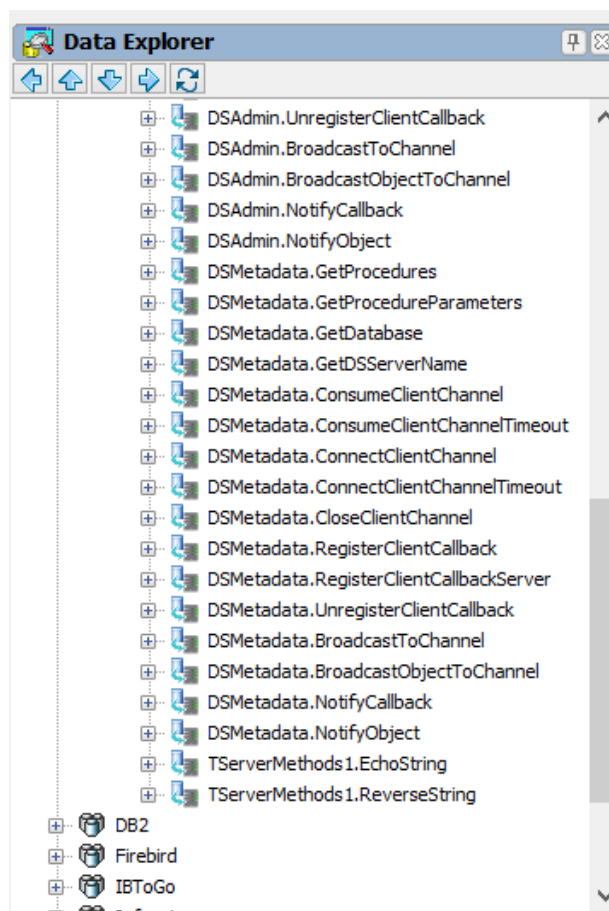
Once we compile it, we can test the sockets and HTTP interfaces from the IDE, using the Data Explorer or using one of the DataSnap client wizards to generate a client application. For now, I will stick to basic testing in the Data Explorer. First, you have to compile and run the server (possibly without debugging so you can keep using the Delphi IDE more freely). Notice, though, that for testing within the IDE, you need to disable the transport filters from the TCP/IP and HTTP connections, or you'll see internal errors.

Next, open the Data Explorer pane (on the far right in the image below), pick the only pre-configured DataSnap connection, use the Modify local menu command, and open the configuration dialog below:



With the test connection button, you can actually try to connect to the running server, and it should succeed. You can replace the http protocol with tcp/ip and switch to port 211 (if you have not changed the defaults) to test a socket connection instead.

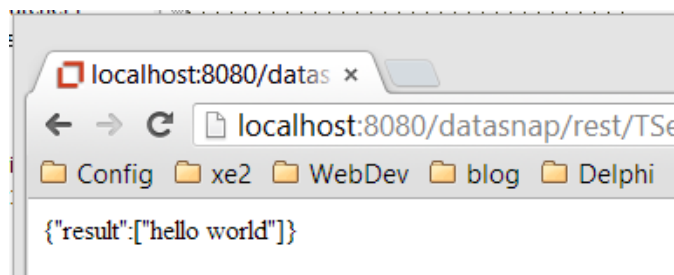
If you now double-click on the node below the connection, ServerMethods, you should see a rather long list of administrative and metadata methods exposed by default by the DataSnap server, this is followed by the two test methods generated by the wizard, EchoString and ReverseString:



Testing the REST interface is much easier, as we can use a standard client capable of making HTTP request... your web browser! Just open any browser and type it a URL like:

`localhost:8080/datasnap/rest/TServerMethods1/EchoString/hello%20world`

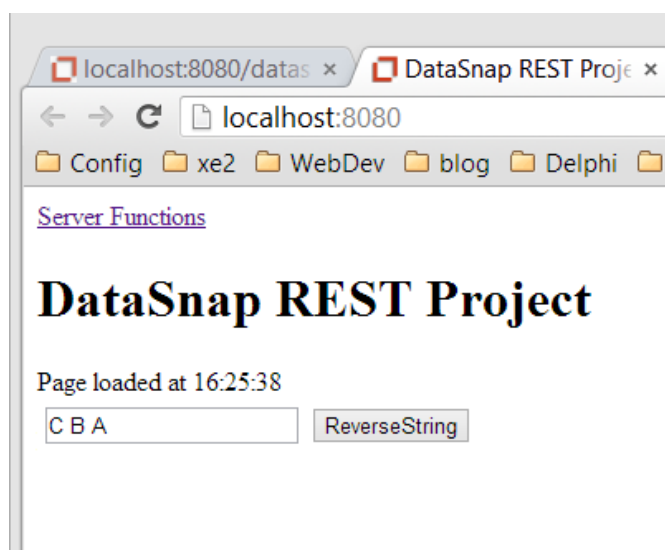
This URL is made of the application address and port (*localhost:8080*), the REST path (*/datasnap/rest*) the name of the class (*TServerMethods1*), the name of the method you want to execute (for example, *EchoString*), and the (optional) parameters of the method. In the URL the %20 is just a replacement for a space, but you can actually type a space in your browser. The result is a JSON response like I've captured here in Chrome:



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). In the specific case, there is only one result, the same string passed as parameter.

As you can imagine, when you compile and build the projects generated by the second wizard you can use a DataSnap HTTP custom Delphi connection or a browser. The WebBroker server also has a standard HTTP response in case you use a URL not starting with /datasnap, but this is a rather dull fixed string I will not even bother showing here.

The situation with the DataSnap REST server is very different, though. Using an URL like the one above you still receive a plain JSON response, but if you ask for the root document you will see a full blown (even if extremely simple) HTML web application:



Given the structure of this application has already been the subject of many sessions and white papers (including those I wrote in the past) I am not going to focus on the various extensions and configuration option, or on how to turn this bare-bone application into a complete and useful project. Rather, I want to focus on deployment.

## DEPLOYING STAND ALONE SERVICES

Stand-alone services are clearly the simplest to deploy. All you need is the Delphi compiled executable file, and optionally some of the extra JavaScript, HTML, or Zip files (for the mobile connectors). You can deploy on an internal company web service for internal consumption, a model particularly suited for socket based connections. You can deploy on internal or hosted servers for public consumption, particularly if you are using HTTP or REST.

To avoid the need of a restart of the DataSnap server application in case you need to reboot the physical server, you might prefer using a service rather than a console or Windows/VCL application.

A side issue to determine the best deployment scenario is to consider where the database server is located, in case you need one (that is going to be true in most case I'd guess). If the database server is internal, an internal hosting might be preferred, given you will not have to worry about security on the connection between the DataSnap server and the database server. You will have to worry about connections between the DataSnap server and the client, but this is an issue you have anyway and you can tackle with either HTTPS or a DataSnap connection encryption filter.

On the other hand, if you prefer hosting the database center in a web farm, using the same physical location for the DataSnap service will generally be a good idea. DataSnap can optimize the network traffic over a direct client/server connection to a remote database server, and also improve the security compared to a database directly exposed to the outside world.

## INTEGRATING WITH WEB SERVERS: LIBRARIES

If you pick the WebBroker model and a HTTP or REST server (not for a socket based one), you can integrate your Delphi application in a web server like Microsoft's IIS. In such a case the DataSnap server is compiled into an ISAPI library, installed and loaded in IIS.

Certainly integrating an application with a web server poses some extra challenges. You often have to learn how to configure the web server, integrate the two, determine which files should better be served statically by the web server and which ones should be returned by the Delphi compiled module.

This model has certainly a few advantages in terms of robustness and security, makes your application rely on the threading model of the web server (often quite optimized) rather than using the threading of the IdHTTPServer or of a similar component. Also, using a web server you can demand of it features like encryption (HTTPS), compression, errors management, and many others.

One of the issues in this model, however, is that given Delphi libraries are compiled, to replace them you might have to end up stopping the web server itself, replace the library, start it up again. In case the physical server hosts many services, this might be really less optimal (although it is now possible in some circumstances to replace a library without stopping the entire service). In the past it was also the case that an error in the library would block the web server, but this does not happen any more due to some constraints in the execution environment for the libraries.

I keep using the plural (web services) for two reasons. First, there are other ISAPI compatible web servers. Second, Delphi has for long time had official support for building Apache modules, the kind of DLLs you load into the Apache web server. This feature was supported for some time by the WebBroker framework, but has now been dropped from Delphi.

## USING A PROXY CONFIGURATION

It is slightly less known that there is another way to let a web server forward a request to a custom application. You build the custom application as a stand-alone secondary web server listening on some odd port, and configure the primary web server (like IIS or Apache) to forward it any request for a given virtual domain, or a given path (or set of paths) of a given virtual hosts, or even use some complex regular expressions to determine which URLs should be mapped to real files and which should be forwarded to a secondary web server or HTTP server applications.

As an example, one of my primary web applications built with the DataSnap REST architecture is running in a web farm on a secondary port and has a corresponding virtual hosts configured in Apache as a proxy:

```
<VirtualHost *:80>
    ServerName mysite.mydomain.it

    ProxyRequests Off
    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>

    ProxyPass / http://localhost:8888/
    ProxyPassReverse / http://localhost:8888/
    ProxyPreserveHost On
</VirtualHost>
```



This is clearly a minimal configuration, applied to an entire virtual host. Incoming requests on port 80 (the standard HTTP protocol port) are redirected to port 8888 of the same computer, hiding this fact to the outside.

You can learn more about Apache proxy configuration (a very complex topic indeed and an extremely long documentation page) at:

[http://httpd.apache.org/docs/2.2/mod/mod\\_proxy.html](http://httpd.apache.org/docs/2.2/mod/mod_proxy.html)

As you can see there, Apache not only offers basic configuration but also failover and load balancing, a topic I'll get back to later on. I only have experience with Apache proxy configuration, but I have been told that proxy support can also be added to IIS.

Why would you want to use a proxy rather than direct web server integration or a stand-alone scenario? Having a web server facing the customers provides advantages in terms of security, activity logging, speed in returning cached files, support for encryption and compression, errors management, and more. They are basically the same reasons for integrating with a web server.

In this scenario, however, you also gain more flexibility. For example, the web server and the DataSnap service can be stopped and restarted independently. The web server will keep responding to request on all other virtual hosts or virtual folders, in case you restart or replace the DataSnap server. And the DataSnap server can stay in memory (with active user session) in case you need to restart the web server.

Also, the web server and the DataSnap server don't need to be installed on the same physical server: Given the two connect over HTTP, they could be on different machines (even with different operating systems) or different virtual machines. For some time I had a Linux host, with Apache installed in it, acting also as a VMware server and hosting a Windows virtual machine, on which I was running a DataSnap server compiled with Delphi. In different scenarios, having two different physical servers can help scale your applications. You can also have a secondary "failover" machine (the third of the pool) waiting idle and acting as a backup in case the first proxy target doesn't respond.

As you might have understood from my words, I like deploying real world servers using the proxy configuration provided by Apache. To me, this offers a great deal of flexibility, power, and control with a minimal configuration.

## MOVING THE DATASNAP SERVER TO THE CLOUD

Rather than hosting the DataSnap server applications or services on a custom physical server in a web farm, you can also host them in a virtual machines living on a cloud. From a technical

point of view, using a hosted solution or one in the cloud makes little practical difference. You still need a Windows machine, need to move an executable there, run it, and configure it.

There is a very important difference though, besides costs structures, speed, availability and all the related considerations shared by other cloud computing scenarios, and this different is the location of your data. Using a virtual machine in the cloud, in fact, you could host the database and file servers on the same machine, but also have dedicated database instance and use a file storage service. This is where a cloud solution becomes unique. In the coming sections, I will introduce some of the cloud-based storage options you have in a Delphi application in general, including a DataSnap server. Later on, I will try putting it all together.

## DATABASES IN THE CLOUD

The term “Cloud Computing” in general implies using some computing services (storage, CPU time, or even complete virtual machines) offered by a web farm along with Internet connectivity. In the early days, having set up a large Web farm, Amazon started selling some extra storage and bandwidth as a side activity, but this is now turning into a big business for Amazon, Microsoft, Google, and most other major Internet players.

As an example, rather than saving a file on your own web server machine, you can host it on a cloud server, which can make the file available to thousands of concurrent users thanks to higher bandwidth and more balanced servers than you can possibly achieve with a single machine or small web farm. Notice that even if the budget is very limited, having your own servers will generally be cheaper... unless you have a usage peak, in which case you would have to invest a lot of money in infrastructure up front. The nice element of cloud computing is you can use it on-demand and it easily scales.

For the sake of our discussion, it is enough to point out to some strengths and weaknesses of the model. Strengths of cloud-based databases include:

- Very easy and fast to set up, including license management if required
- Very scalable, given the “infinite” space and bandwidth offered, easy and very fast to adapt to changing needs
- Fixed and relatively simple cost structure

This is a very good approach (and in most cases the only one) when you want to deploy middle-tier servers (like DataSnap servers) on cloud computers, like Amazon's EC2. If the same web farm hosts both your database and your server application instances, the speed between them will likely be similar to a very fast internal network.

Some of the disadvantages of this approach include:

- Costs often much higher than native hosting, if requirements are stable over time (that is, if you are not envisioning usage peaks)
- Slow compared to an internal solution, if the users are from inside the company

Now there are basically two sides in terms of database offerings in the cloud, as you can use either relational databases or NOSQL databases.

## RELATIONAL DATABASES IN THE CLOUD

Amazon Web Services (AWS) and Microsoft's Azure both offer relational databases in the cloud. There are certainly many other vendors with similar opportunities, but these two clearly stand out. These databases are available from these two companies:

- **MySQL**, hosted on Amazon Web Services – it was the first of these offers and dates back a few years and it is one of the cheapest solutions.
- **Oracle Database**, hosted on Amazon Web Services and clearly a bit more expensive than the MySQL offer.
- **SQL Server**, not surprisingly the only database hosted on Microsoft's Azure.

Now the easy element for Delphi developers (and other developers in general) is that you connect to these databases in the same way you do for client/server ones. You need a data access component (for example dbExpress), the client library, and a specific connection string referring to the proper domain or IP address. I've personally tested connections to SQL Server on Azure and MySQL on AWS with success.

As an example, these are the parameters of a TSQLConnection I used to access to an SQL Azure database with a connection like:

```
object MyAzure: TSQLConnection
  ConnectionName = 'MyAzure'
  DriverName = 'MSSQL'
  GetDriverFunc = 'getSQLDriverMSSQL'
  Params.Strings = (
    'drivername=MSSQL'
    'HostName=x41fer9cqm.database.windows.net'
    'Database=firstsample'
    'User_Name=...'
    'Password=...' )
end
```

## USING SPECIFIC CLOUD SERVICES FROM DELPHI

Delphi has a specific library for cloud services. This portion of the Delphi library has two visual components for holding connection information (like the account name or ID and a password):

- *TAmazonConnectionInfo* is for Amazon's AWS services
- *TAzureConnectionInfo* is for Microsoft's Azure services



The actual cloud service classes take these connection components as constructor parameters. The `Data.Cloud.AmazonAPI` unit defines the following classes:

- *TAmazonStorageService* is a mapper for Amazon's S3, Simple Storage Service
- *TAmazonTableService* is a mapper for Amazon's SimpleDB API (a NoSQL database)
- *TAmazonQueueService* is a class for using Amazon's Queue Service

In parallel, Azure support is now in the unit `Data.Cloud.AzureAPI` (while the Delphi XE Azure units still exists mostly for backwards compatibility):

- *TAzureTableService* is used to access Azure's Table service (a NOSQL database similar to SimpleDB, not to be confused with Azure SQL Server service, which offers Microsoft's relational database in a cloud configuration)
- *TAzureQueueService* is a class interfacing the Queue service
- *TAzureBlobService* is the service for binary or file storage, like S3.

Here is a summary of the names of the services for the two offerings supported by Delphi XE3:

	 Windows Azure™	 amazon web services™
Storage	Azure Blobs Service	Amazon S3
Table	Azure Table Service	SimpleDB
Queue	Azure Queue Service	Amazon SQS

While the various services share a base abstract class, called *TCloudService* and defined in the unit `Data.Cloud.CloudAPI`, this is more of an infrastructure class, as there are too many differences among the services themselves to have a single base class with the same virtual methods for, say, listing, getting, and uploading files.

This is the class hierarchy for Cloud Services in Delphi:

```
TCloudService
  TAmazonService
    TAmazonStorageService
    TAmazonBasicService
    TAmazonTableService
    TAmazonQueueService
  TAzureService
    TAzureTableService
    TAzureQueueService
    TAzureBlobService
```

## NOSQL DATABASES IN THE CLOUD

As we have seen Delphi includes some components for Azure and AWS. Using these ready to use classes you can easily populate tables and retrieve data. Or you can read and write BLOBS, which often host images but can also be used to store data in JSON, XML, or any other custom format. Even the CDS format, if you like.

In the following snippets I'll focus only on using the two Column-based NOSQL databases, that is Tables (Azure) and Simple DB (AWS). For example, you can get all of the entries in an Azure table by calling:

```
var
  rowsList: TList<TCloudTableRow>;
  aRow: TCloudTableRow;
begin
  rowsList := TableService.QueryEntities(tablename);
  for aRow in rowsList do
    ...
```

You can also query for one specific row and access some of the columns (or fields) of that row with calls like:

```
var
  aRow: TCloudTableRow;
begin
  aRow := TableService.QueryEntity(tablename, rowKey, partitionkey);
  meDescription.Text := aRow.GetColumn('description').Value;
```

The Rowkey and PartitionKey is two values uniquely identifying each row. You can also pass a FilterExpression parameter to filter rows with specific values. Filtering is even easier when you use AWS tables, as there is a SelectRowsXML function that basically accepts SQL-like

statements, as “select \* from <tablename>”, and accepts *WHERE*, *LIKE*, *ORDER BY*, and *LIMIT* clauses.

## PART III:

# MAKING SERVICES SCALABLE AND ROBUST

If you have followed me up to this point, you now know that DataSnap servers come with different internal architectures, different connection models, and different deployment options. While for an internal server using sockets and keeping an active TCP/IP connection between each client and the server makes individual operations fast, this is far from optimal in case of unstable connections or in case the number of clients grows.

The HTTP stateless model requires you to set a connection for each request, and serve it using a specific thread, and this certainly takes time. The advantage, though, is that while a client is not making a request, it doesn't need to be connected and this lets the server handle a larger number of requests. The total number of connections is the total number of concurrent requests, not the total number of clients. At the same time, using server-side sessions, a DataSnap server can manage specific data for each user, handle authentication and authorization, and track the application status, all with a rather lightweight model. In other words, only the HTTP-based and even more the REST-based DataSnap servers should be more scalable and flexible than the socket-based ones, particularly for Internet development.

## TUNING THE DATASNAP REST SERVER

If you use the wizard to create a DataSnap REST server, you might expect to get a project with the best possible configuration. This is not the case, however. What the wizard generates is a rather standard code, not particularly optimized. To make things worse, if you try to test the server by making as many calls as you can, you end up causing unwanted side effects, like generating a new session for each request.

I have written a standard server and a simple testing client to show this scenario... and see what we can do to improve the speed and the throughput of a DataSnap server. The initial server project is what you get out of the DataSnap REST Application wizard, with the DSServer component on its own data module to avoid creating it over and over. The project used for testing is a simple VCL application with a testing loop like the following (placed in a thread to avoid blocking the client user interface and listed here with some omissions):

```
procedure TextThread.Execute;  
var  
    sw: TStopwatch;  
    I: Integer;
```

```
    strUrl: string;
    IdHTTP1: TIDHTTP;
begin
    IdHTTP1 := TIDHTTP.Create(nil);
    try try
        strUrl := 'http://127.0.0.1:8080/datasnap/rest/' +
            'TServerMethods1/ReverseString/Hello';
        sw := TStopwatch.StartNew;
        for I := 1 to nInteractions do
            begin
                IdHTTP1.Get(strUrl);
            end;
        sw.Stop;
    except on E: Exception do
        Form7.Memo1.Lines.Add (IntToStr (ThreadId) +
            ': Exception ' + E.Message);
    end;
    finally
        idhttp1.Free;
    end;
    Form7.Memo1.Lines.Add (IntToStr (ThreadId) + ':' +
        FloatToStr (nInteractions * 1000 / sw.ElapsedMilliseconds));
end;
```

The output of the program shows the number of requests per second that the server could handle, and of course the numbers change a lot depending on the server and client hardware, the bandwidth between the two, and other considerations. In my specific case, I am using a single computer running both server and client, which is not an optimal configuration, but it is certainly easy to reproduce over time. If I run this code against a plain server my DataSnap server runs about 500 requests a second (495 on average).

There are two side effects a developer could notice. First, the server creates and destroys a large number of threads (basically one for each request). Second, the server seems to use more and more memory, although there is no memory leak reported.

Regarding threading, creating one for each incoming request is Indy's IdHTTPServer default configuration, but you can tune it adding code to the server main form, which creates and manages the Web server component. The OnCreate event handler of this form initializes the HTTP server, with the code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    FServer := TIdHTTPWebBrokerBridge.Create(Self);
```

```
end;
```

Now we can change the configuration to use a thread pool, pre-allocating a number of threads for the incoming concurrent requests:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  SchedulerOfThreadPool: TIdSchedulerOfThreadPool;
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  SchedulerOfThreadPool := TIdSchedulerOfThreadPool.Create(FServer);
  SchedulerOfThreadPool.PoolSize := 50;
  FServer.Scheduler := SchedulerOfThreadPool;
  FServer.MaxConnections := 50;
end;
```

As you can see in the last line above you can also put a limit to the concurrent connection (MaxConnections): After reaching this limit the server will stop responding and simply return an error. This value should probably be quite high, but it is relevant, as it will prevent the server from shutting down in case of an attack or simply an excessive workload. It is often better to return a clear error message (“too many connections”) than simply fail to respond under the load.

While with this code we reduce the load on the server in terms of thread creation, the speed increase under my testing scenario (single computer) is negligible. On a heavier load and with many connections, things will vary.

The second issue is the apparent memory consumption. This is just the effect of the sessions begin created on the DataSnap server and being kept in memory for 20 minutes. While you can change the session timeout, this won't have a direct effect. The issue here is that the testing code making consecutive calls should do so within a single session. A technical solution is to simulate what Delphi's REST connections do: use extra headers to convey session information.

What I have added to my testing code is a first call to create and return the session information, which is then copied to HTTP extra headers. Here are the lines I added before the calling loop:

```
IdHTTP1.Get(strUrl);
strSession := Copy(
  IdHTTP1.Response.RawHeaders.Values ['Pragma'], 1, 30);
IdHTTP1.Request.CustomHeaders.Clear;
```



```
IdHTTP1.Request.CustomHeaders.AddValue('Pragma', strSession);
```

If you want to make sure this code is effective you can look at the server memory consumption, but also do a direct test adding this code to the server application:

```
ShowMessage(IntToStr(TDSSessionManager.Instance.GetSessionCount));
```

This second change keeps memory consumption under control in the testing scenario, but again does not really affect the throughput. Thread pooling and sessions management can certainly affect performance in real world situations, but DataSnap in itself has quite an overhead. In other words, DataSnap offers a rather sophisticated call processing layer, matching methods and parameters, performing conversions to and from JSON, and more. This layer has some runtime overhead, of course.

A way to measure it is to add a plain response to the WebBroker server, defining a new custom action. My new action is configured as follows:

```
item
  Name = 'WebActionItem1'
  PathInfo = '/direct'
  OnAction = webModule1WebActionItem1Action
End
```

The code of this action could be very simple like:

```
procedure TwebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '{"result": "hello world"}';
end;
```

You can execute this code from the client using (in my configuration) an URL like:

```
http://127.0.0.1:8080/direct
```

By repeating this direct WebBroker call rather than a DataSnap call my throughput grows from around 500 requests a second to 50% more, to values around 750 requests a second. This is a measure of the execution overhead of a DataSnap request compared to a WebBroker one. If this sounds like a lot of extra time, consider that these are basically do-nothing operations. In case the DataSnap server has to make a database call and do some real data processing, the call overhead will become a fraction of the actual execution time.

## PROXIES, FAIL OVER, LOAD BALANCING, AND SESSIONS

Now, suppose you want your DataSnap server to really scale and survive system crashes. A classic way to achieve this (on your own hardware or on a cloud service) is to have multiple physical or virtual servers with the same configuration and running services. If you use a specific proxy service (or Apache or IIS, as mentioned earlier) you can redirect the traffic to a single computer. In case this computer doesn't respond the traffic gets redirected to a new one. This is a scenario that a Delphi DataSnap server can handle.

What is even nicer, though, is the ability to use a proxy (or some network configuration engine) to send part of the requests to a server and some other requests to a second running server. In the most naïve scenario, 50% of the requests goes to each, but more sophisticated tools (again, Apache proxy can do this) will distribute the requests depending on the average response time, balancing the effective load among two or more servers.

The question now becomes if we can use a DataSnap REST server in such a scenario. Given that the architecture is completely stateless and you can have your database server on separate hardware (or on a virtual cloud database), can you let requests follow different paths and reach different DataSnap servers? Unluckily this would only work for the simplest applications, because DataSnap servers keep session information in memory (including login status, permission, and so on). If your application requires no real login and session data, you could be able to deploy your servers following this path. This might be true for a "pure service" for which you handle permissions in a custom way (for example using extra HTTP headers for each request).

However, if you need even minimal session and login management, moving session information out of memory and into a shared database (or a shared service) is not exactly a trivial operation to do in a DataSnap server. This might be an interesting feature to add in the future.

## PART IV:

## ACCESSING APPLICATION SERVICES FROM DIFFERENT TYPES OF CLIENTS

Now that I have covered deployment and discussed some configurations that can help scale your DataSnap server application, in the final part of this document I want to provide a short overview of different types of clients that you can build for a DataSnap server application. Again, my primary focus is on pure REST services, given their increased flexibility and scalability.

## CALLING THE REST SERVER FROM A DELPHI CLIENT

Earlier in this paper I built a server and have shown how to use the Database Explorer or a web browser to reach it. Now we can see how to write a Delphi client application for this server. We can use two different approaches.

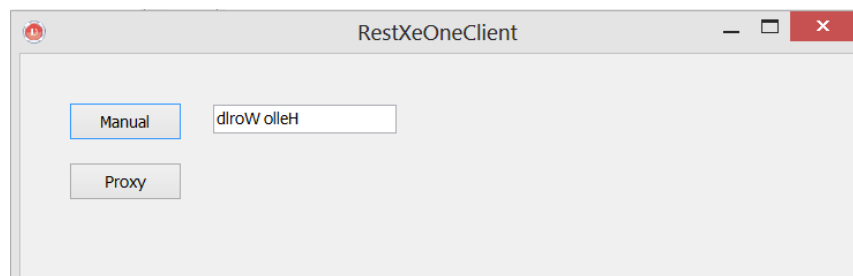
The first approach is to write a Delphi DataSnap client, using the specific REST client support. The second approach is to create a custom REST client and use the IdHTTP component to call the server and the DBXJSON support unit to parse the result. Here I will use both techniques, starting with the “manual” one and later getting to the automatic support, which should be generally preferred.

First, I have 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 have 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;
```

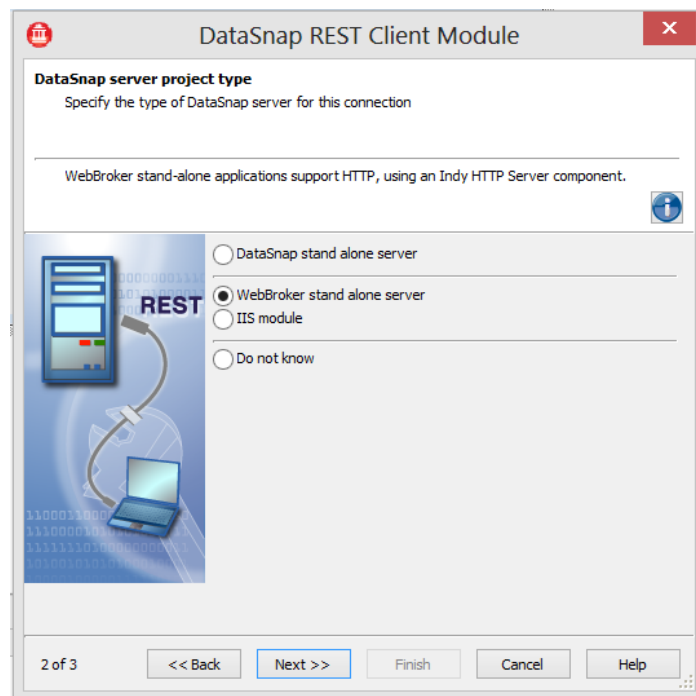
This call builds a proper URL by concatenating the server address, the relative path to reach the given method within the REST server, and the only parameter. The final code of the demo does the full parsing of the resulting JSON string, copying the result back to the edit box, so if you enter the text *“Hello World”*, the call results in the following output:



## 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 DataSnap REST Client Module Wizard. 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 would 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 have 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 do not 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:

```
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 exactly the same 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 a 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 HTML AND JAVASCRIPT SAMPLE CLIENT

As I have already shown earlier, when you create a DataSnap REST Application you end up with a server, but also with a HTML and JavaScript sample client, powered by the same server (using its WebBroker features).

The HTML of the sample page (if we ignore the extra code used to manage login and described later) is quite simple:

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

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.

## THE JQUERY MOBILE CLIENT

Given a DataSnap REST server, you can use JavaScript to build a mobile client application, rather than a standard browser based one. The difference between the two models is in the form factor, but also in the user interface controls, which in a mobile client tend to mimic those of the mobile platform (generally iOS).

I have built some simple mobile clients using jQuery Mobile (<http://jquerymobile.com>), an extension to the code jQuery JavaScript library I will introduce more details on the session covering jQuery. Here I will just summarize a couple of key elements of an example. My application has a single HTML file but it is made of three separate pages, alternatively shown (here I will cover only the first page, though). The page links with special CSS styles to create buttons and to turn HTML lists into controls like those on your phone and to make them touch-friendly.

For example the top part of the main page is defined with some rather plain HTML:

```
<div data-role="page" id="main_page">
  <div data-role="header">
    <h1>DataSnap REST Demo</h1>
  </div><!-- /header -->

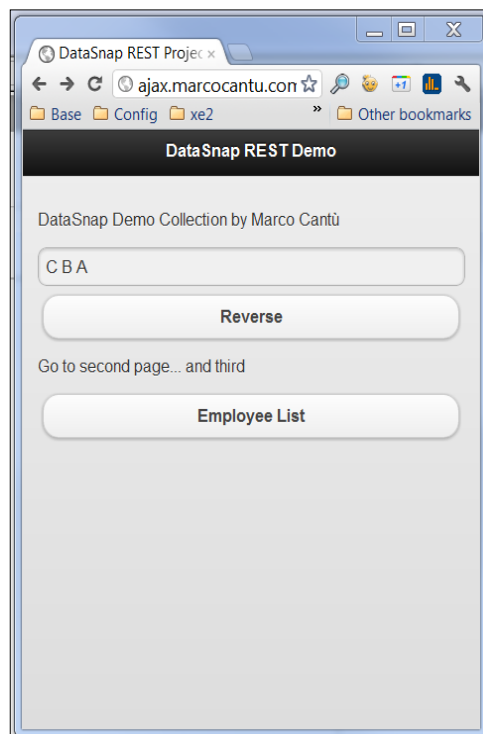
  <div data-role="content" id="content">
    <p>DataSnap Demo Collection by Marco Cantù</p>

    <input id="valueField" type="text" value="A B C" />
    <a href="#" data-role="button" id="reverse">Reverse</a>

    <p>Go to second page... and third</p>

    <a href="#list_page" data-role="button" id="listpage">
      Employee List</a>
  </div><!-- /content -->
</div><!-- /page -->
```

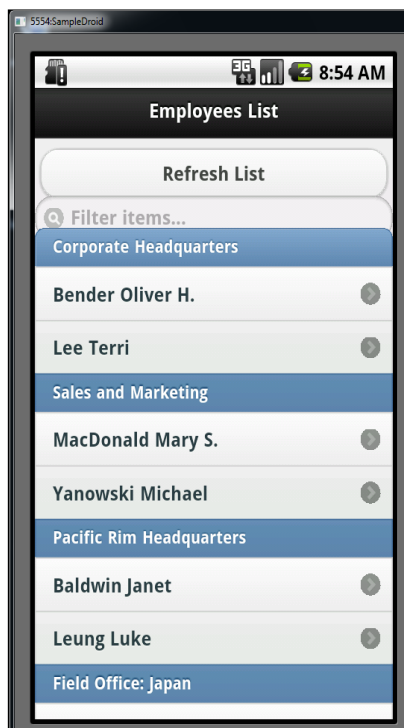
The output of this page can be seen on a regular PC browser, provided you re-size it properly, but it really makes sense on mobile devices:



The reverse string button (which is technically not a button, but looks like one) use some standard jQuery code:

```
$('#reverse').click (function (){  
    var value = $('#valueField').val();  
    var value = serverMethods().ReverseString(value).result;  
    $('#valueField').val(value);  
});
```

The other two pages use the same approach for showing the list of employees (which can be filtered on the client) and the individual employee data. Again, the HTML is pretty simple and the JavaScript processing the page is not much different from what you will use in a standard HTML application. Here is the output of the second page, as example (I will not list the code here):



## BUILDING AN ANDROID WEB APPLICATION

The simple the jQuery Mobile client covered earlier can be transformed into a Web Android Application. The application only has a full screen Web browser control, and behaves exactly like the mobile web page. However, you can activate it with the icon (I know, I have been lazy



and left in the default Android application icon!) and it will show at full screen, that is, without the browser address bar and other browser elements.

In terms of code, to build such an Android application you have to use the standard Android development tool set based on Eclipse, and write your code in Java. Do not worry, as the code required for this demo is extremely simple.

The application has a form made of a WebView control, like the following:

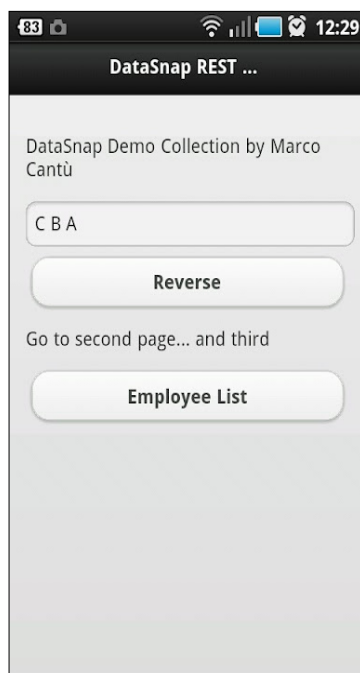
```
<?xml version="1.0" encoding="utf-8"?>
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:scrollbars="none" android:id="@+id/webview"/>
```

As the program starts, the WebView loads the proper URL:

```
public class mcjqmobile extends Activity {
    @Override

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //Remove title bar as we already have it in the web app
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);
        //Point to the content view defined in XML
        setContentView(R.layout.main);
        //Configure the webview setup in the xml layout
        WebView myWebView = (WebView) findViewById(R.id.webview);
        WebSettings webSettings = myWebView.getSettings();
        //Yes, we want javascript, pls.
        webSettings.setJavaScriptEnabled(true);
        //Make sure links in the webview is handled by the webview
        // and not sent to a full browser
        myWebView.setWebViewClient(new WebViewClient());
        //And let the fun begin
        myWebView.loadUrl("http://ajax.marcocantu.com/mobiledsnap/");
    }
}
```

That is all the code for the application. Compile it and make the APK file available as a download or publish in an Android market, you are ready to go. Here is the first page, similar (but not identical) to the browser-based version shown earlier:



## DATA SNAP MOBILE CONNECTORS (FOR ANDROID)

If your goal is to build a mobile application, using the native tools, there is another approach you can follow. DataSnap REST Applications (but also other DataSnap servers) have the ability to generate proxy interfaces in languages other than Delphi, C++, or JavaScript. This technology is called “mobile connectors” and helps on two different counts:

- It generates a proxy file with a class in the given language (C#, Java, or ObjectiveC) mapped to the REST methods exposed by your server.
- It offers extensive support for managing server connections, parsing JSON, mapping data types, and handling callbacks. In other words, you have a set of basic classes in each of these languages for helping you build REST clients and call into the Delphi DataSnap REST server.

As I mentioned, there are several platforms supported, mostly the mobile platforms, here listed after the token used to refer to them:

- **java\_android**: Android 2.1
- **java\_blackberry**: BlackBerry SDK 5 and 6
- **csharp\_silverlight**: Windows Phone 7
- **objective\_ios42**: iOS 4.2 (but there is also support for iOS 5.0)

The additional option introduced in Delphi XE2 Update 4 was:

- **freepascal\_ios50:** FireMonkey clients compiled in FreePascal using XCode for the latest version of iOS (but there is also support for iOS 4.2)

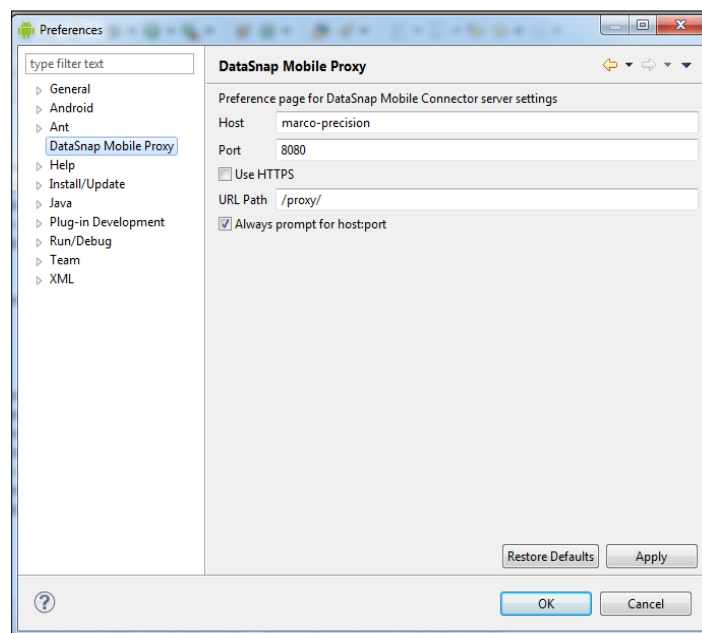
The support is provided by the TDSProxyGenerator generator component. You can retrieve a ZIP file with the source code by connecting to the application URL using the path:

`http://<site>/proxy/<token>.zip`

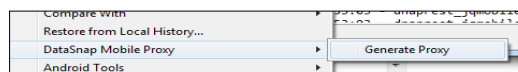
As an alternative, there is a command-line utility, called Win32ProxyDownloader. Again, you can run it against a compiled application, for example typing:

```
win32ProxyDownloader -language java_android -host localhost:8080
```

If you use Eclipse for Android development, there is even a plug-in of the IDE that lets you retrieve the proper source code files and update them right into the Eclipse IDE. You can see the configuration of this plug-in in the Eclipse Preferences:



To invoke it, select the project menu and use the corresponding command:



## AN ANDROID JAVA SAMPLE APPLICATION

Equipped with his information, I have proceeded building a very simple native Android client for the Delphi DataSnap Rest server in Eclipse. First of all, I have downloaded the proxy interfaces and helper classes. The ZIP file you receive has the three nested folders `com\embarcadero\javaandroid` (given that folders structures match name spaces in Java) with 56 support files for the various data types involved, plus the a `DSProxy.java` source code file, including the Java proxy class declaration (from which I have omitted the private declarations for the commands metadata):

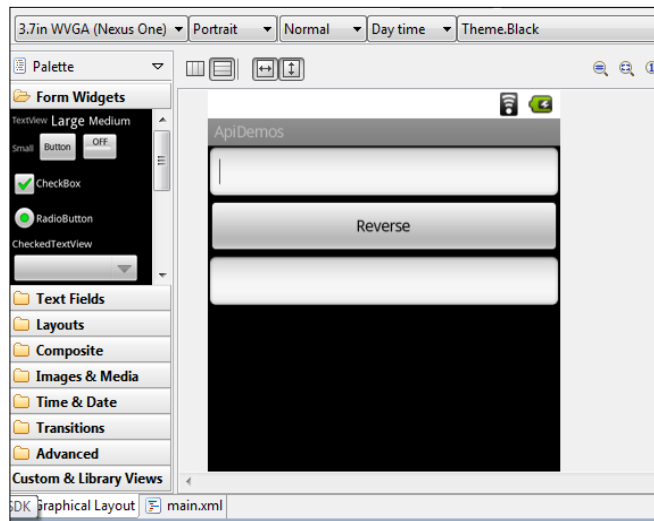
```
package com.embarcadero.javaandroid;

import java.util.Date;

public class DSProxy {
    public static class TServerMethods1 extends DSAdmin {
        public TServerMethods1(DSRESTConnection Connection) {
            super(Connection);
        }

        /**
         * @param Value [in] - Type on server: string
         * @return result - Type on server: string
         */
        public String EchoString(String Value) throws DBXException {
            DSRESTCommand cmd = getConnection().CreateCommand();
            cmd.setRequestType(DSHTTPRequestType.GET);
            cmd.setText("TServerMethods1.EchoString");
            cmd.prepare(get_TServerMethods1_EchoString_Metadata());
            cmd.getParameter(0).getValue().SetAsString(Value);
            getConnection().execute(cmd);
            return cmd.getParameter(1).getValue().GetAsString();
        }
        ...
    }
}
```

For the client application, I created a simple form, designed like the following:



Here is the corresponding XML description of the visual controls:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">
    <EditText android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/editText1">
        <requestFocus></requestFocus>
    </EditText>
    <Button android:id="@+id/button1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/Reverse"></Button>
    <EditText android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/editText2"></EditText>
</LinearLayout>
```

As you can see, there are basically two edit boxes and a button. These are the components our application will interact with, in the code behind this main form:

```
public class Reverse2Activity extends Activity {
    /** Called when the activity is first created. */
    @Override
```

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Button button = (Button) findViewById(R.id.button1);
    button.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            EditText tedit = (EditText) findViewById(R.id.editText1);
            EditText tedit2 = (EditText) findViewById(R.id.editText2);

            DSRESTConnection conn = new DSRESTConnection();
            conn.setHost("192.168.1.161");
            conn.setPort(8080);
            TServerMethods1 proxy = new TServerMethods1(conn);
            try {
                tedit2.setText(
                    proxy.ReverseString(tedit.getText().toString()));
            } catch (DBXException e) {
                e.printStackTrace();
            }
        }
    });
}
```

The code installs an event handler for the button (using a closure or anonymous method). In this method the program initializes the REST connection, connects to the proxy, and calls the `ReverseString` method using the text of the first edit as input and copying the result in the second edit box. Nothing fancy, but should give you an idea of the process involved.

## ASSESSMENT: DATASNAP USAGE SCENARIOS

We have seen in the initial part of this paper that DataSnap accounts for many different types of servers and connectivity. In this last part we have seen that even client applications can vary significantly, from classic Delphi applications to web and mobile clients. These last two options, though, are limited to DataSnap servers with the REST interface (whether they are classic servers or based on WebBroker).

In other words, if your goal is an internal multi-tier architecture with Delphi or C++Builder clients you can use sockets and HTTP and take advantage of the dual interfaces (datasets exposed by providers, plus remote methods invocation). If your goal is to deploy an open and scalable architecture, independent from the client side technologies, you should probably stick with REST and the methods invocation only.

DataSnap offers a solid and open foundation, which Embarcadero plans expanding in the future. In fact, given that Delphi is becoming a multi-platform and multi-device development environment, with direct support for mobile clients, the need for a flexible server-side data access solution is going to grow.

## ABOUT THE AUTHOR

Marco Cantù recently joined Embarcadero Technologies as Delphi Product Manager. He was 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 (from 2007 to XE).

Marco is a frequent conference speaker, author of countless articles on Delphi, and used to teach advanced Delphi classes (including Web development with Delphi) at companies worldwide. You can read Marco's blog at <http://blog.marcocantu.com>, follow him on Twitter as @marcocantu, and contact him on [marco.cantu@embarcadero.com](mailto:marco.cantu@embarcadero.com).

## ABOUT EMBARCADERO TECHNOLOGIES

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).