

White Paper

---

# Delphi 2010 における REST

Marco Cantù (<http://blog.marcocantu.com>)

2009 年 11 月

---

## 要旨

REST (Representational State Transfer) は、業界に重大な影響を及ぼしつつある新しい Web サービスアーキテクチャです。大手のベンダ (Google、Yahoo、Amazon、Microsoft) から提供される新しい公開 Web サービスの大半は、複数の情報源から得られる情報を共有および結合するための技術として REST を利用しています。

REST アーキテクチャを実装するという事は (HTTP や XML のような) シンプルな技術だけを用いるということになるため、Delphi ではこれまで REST を十分サポートしてきました。現在、バージョン 2010 には、DataSnap インフラストラクチャの一部である REST サーバーの開発を対象としたサポート機能が追加されています。

このホワイトペーパーでは、REST に関与する技術を Delphi の観点から詳細に調べ、普及している Web サイトのクライアントアプリケーションを作成する方法と、Delphi 2010 に用意されているサポート機能を用いて REST サーバーを作成する方法を示します。

## はじめに

ここ 10 年にわたって Web の爆発的普及を目の当たりにしてきましたが、現在はいわゆる Web 2.0 がそうした状況にあります。完全には把握しづらいことが多い世界的規模の相互接続の中で、異なる Web サイト間、Web サイトとクライアントアプリケーションの間、Web サイトとビジネスデータベースの間の自動的なやり取りが姿を現し始めたばかりです。

Web 上では、人がブラウズできる以上の速さでデータが移動するため、売上データ、財務情報、オンラインコミュニティ、マーケティングキャンペーンなどのさまざまな情報源から得られる情報の検索、追跡、監視を行えるプログラムが強く求められています。

## なぜ WEB サービスなのか

急速に頭角を現しつつある Web サービス技術は、ビジネスへのインターネットの役立ち方を変える可能性を秘めています。Web ページをブラウズして注文を入力するのは個人つまり B2C (企業・消費者間取引) アプリケーションの場合にはかまいませんが、企業つまり B2B (企業間取引) アプリケーションの場合には良くありません。本を 2、3 冊買いたいのなら、書店の Web サイトにアクセスして注文を入力するだけでいいかもしれません。しかし、書店を経営していて 1 日に何百もの注文を出すのであれば、これは効率的なアプローチとは程遠く、売上の追跡と追加注文の決定に役立つプログラムを持っている場合は特にそうです。このプログラムの出力を取得して別のアプリケーションに入力し直すのはばかげています。

Web サービスはこのような問題を解決するためのものです (もっと正確に言えば、本来はそのためのものでした)。売上の追跡に使用されるプログラム側では自動的に要求を生成して Web サービスに送信でき、Web サービス側ではその注文に関する情報を直ちに返すことができます。次のステップは、たとえば、発送品の追跡番号を要求することかもしれません。この時点で、プログラムは別の Web サービスを使用して、送り先に届くまで発送品を追跡できるため、いつまで待たされるかを顧客に知らせることができます。発送品が届くと、プログラムでは、注文が確定していない人々に SMS、ポケットベル、あるいは Twitter で督促状を送付でき、銀行の Web サービスで支払指図書を

発行できます（例はまだ先がありますが、ここまでで考え方はおわかりいただけたと思います）。Web と電子メールでは人がやり取りできますが、Web サービスはコンピュータの相互運用性のために作られています。

Web サービスの話題は幅広く、数多くの技術とビジネス関連の標準規格が関係してきます。ここでは、全体像とビジネスへの影響を論じるのではなく、いつものように、ベースとなる Delphi 実装と Web サービスの技術面を重点的に扱います。Delphi for Win32 には、かなり高度な Web サービス サポート機能が用意されています。これは当初 SOAP の形で導入されたもので、現在では HTTP コンポーネントと REST を用いて容易に拡張できます。

## WEB サービス技術: SOAP と REST

Web サービスの考え方はかなり抽象的です。技術に関して言えば、開発者を引きつけている主なソリューションが現在 2 つあります。1 つは SOAP（Simple Object Access Protocol）標準（<http://www.w3.org/TR/soap/> を参照）を用いること、もう 1 つは REST（Representational State Transfer）アプローチをそのバリエーションの 1 つである XML-RPC（XML-Remote Procedure Call）と共に用いることです。

注意すべきなのは、どちらのソリューションも通常は、伝送プロトコルとして HTTP を使用し（ただし、代替手段も提供されています）、データのやり取りに XML（または JSON）を使用する点です。標準 HTTP を使用することで、Web サーバーが要求を処理でき、関係するデータ パケットがファイアウォールを通過できます。

このホワイト ペーパーでは、SOAP の詳細には触れず、REST だけを重点的に扱います。まず理論的基礎を少し説明したあと、サーバーおよびクライアントの "手製の" 簡単な例を示し、普及している REST Web サービスの REST クライアントの開発について深く考察し、最後に、DataSnap アーキテクチャの拡張機能として、Delphi 2010 で利用可能な REST サーバー側サポートに焦点を合わせます。

## REST(REPRESENTATIONAL STATE TRANSFER)の 背後にある概念

たとえ REST の概念がこのところ流行しているとしても、この改まった名称の導入とその背後にある理論の提唱はかなり最近のことです。なお、前もって言うておかなければなりません、REST の正式な標準は存在しません。

REST（Representational State Transfer）という用語はもともと Roy Fielding 氏が自身の博士論文（2000 年）で考案したもので、SOAP 標準に頼らず HTTP と URL を用いて Web でデータにアクセスすることの代名詞として急速に広まりました。

REST という用語は当初、Web ブラウザとサーバーとの関係を記述したアーキテクチャ スタイルを表すために用いられました。（ブラウザが特定のクライアント アプリケーションのどちらかを使って）Web リソースにアクセスすると、サーバーはリソース（HTML ページ、画像、何かの未処理データなど）の表現をユーザーに送信するという考え方です。その表現を受信するクライアントは所定

の状態に設定されます。クライアントが（たぶんリンクを使って）さらに情報やページにアクセスすると、クライアントの状態は変化し、前の状態から遷移します。

Roy Fielding 氏はこう解説しています。

「"Representational State Transfer" は、うまく設計された Web アプリケーションの動作つまり Web ページのネットワーク（仮想的なステート マシン）のイメージを連想させることを意図したもので、そこでは、ユーザーはリンクを選択することでアプリケーション内を移動し（状態遷移）、その結果、次のページ（アプリケーションの次の状態を表す）がユーザーに転送されて表示され、使用できるようになります。」

## REST アーキテクチャの要点

したがって、REST はアーキテクチャ（あるいは、アーキテクチャ スタイルと言った方が適切かもしれませんが）であるとしても、標準でないことは明かです。ただし、HTTP、URL、実データのさまざまな形式といったいくつかの既存標準を使用しています。

SOAP は HTTP と XML に準拠しそれらに基づいて構築されているのに対して、REST アーキテクチャでは、以下のように、HTTP と XML（あるいはその他の形式）をそのまま使用しています。

- REST では URL を使用してサーバー上のリソースを特定します（これに対して、SOAP ではさまざまな要求に対してただ 1 つの URL を使用し、要求の詳細は SOAP エンベロープに記述されます）。URL を使用してリソースに対する操作ではなくリソースそのものを特定するという考え方であることに注意してください。
- REST では HTTP メソッドを使用して、実行する操作を指定します（取得の場合は HTTP GET、作成の場合は HTTP PUT、更新の場合は HTTP POST、削除の場合は HTTP DELETE）。
- REST では HTTP パラメータを（クエリ パラメータにも POST パラメータにも）使用して、詳細な情報をサーバーに提供します。
- REST では認証、暗号化、セキュリティに HTTP (HTTPS) を利用します。
- REST では、複数の MIME 形式 (XML、JSON、画像など多数) を使って、データをテキスト形式のドキュメントとして返します。

この種のシナリオには、考慮に値するアーキテクチャ要素がかなりたくさんあります。REST では、システムに対して次のような要件があります。

- 事実上、クライアント/サーバーであること（ここではデータベースの RDBMS とは直接には無関係です）。
- 本質的にステートレスであること。
- キャッシュに対応しており（サーバー側のデータが変更されない限り、同じ URL が順に 2 回呼び出された場合は同じデータを返さなければならない）、プロキシ サーバーとキャッシュ サーバーをクライアントとサーバーの間に挿入できること。その結果として、当然ながら、GET 操作はすべて副作用がないことが必要です。

REST の理論には、この短いセクションで扱った内容よりもはるかに多くの事柄が確かにありますが、ここまでの説明でもその理論の初歩は理解していただけたと思います。このあと出てくる実例と Delphi コードをご覧になれば、主な概念がはっきりするはずです。

## REST 技術と DELPHI

REST 標準というものはなく、REST 開発に特定のツールを使用する必要はないとは言っても、REST が準拠していて、ここで手短かに紹介しておく価値のある既存標準はあります（おのおの詳細な説明をしようとすれば、本が 1 冊できてしまいますので、ここでは割愛させていただきます）。ここでは、これらの技術に対する Delphi でのサポートに特に焦点を合わせます。

### HTTP(クライアントおよびサーバー)

HTTP (HyperText Transfer Protocol) は、WWW (World Wide Web) の中核をなす標準であり、説明の必要もないでしょう。とは言え、1 点だけ触れておきましょう。HTTP は Web ブラウザで使えますが、他のいかなるアプリケーションでも使えます。

Delphi アプリケーションで、HTTP を使用するクライアント アプリケーションを作成する最も簡単な方法は、Indy HTTP クライアント コンポーネントつまり `IdHttp` を利用することです。URL をパラメータとしてこのコンポーネントの `Get` メソッドを呼び出すと、任意の Web ページと多くの REST サーバーのコンテンツを取得できます。時には、他のプロパティを設定して認証情報を指定したり、SSL サポート用の 2 番目のコンポーネントをアタッチする必要があるかもしれません（後で例をいくつか示します）。このコンポーネントでは、`Get` 以外にもさまざまな HTTP メソッドをサポートしています。

Indy スイートではブロッキング スレッドを使用する、つまり、要求の結果が返ってくるまでプログラムのユーザー インターフェイスが動かなくなる（低速の Web サーバーや大量のデータ転送の場合には長い時間がかかります）ので、念のために `IdHttp` 要求は一般にスレッドの中で発行したほうがよいことに注意してください。このホワイト ペーパーに登場するデモでは概してスレッドを使用しませんが、これはプログラムを簡単にするためにすぎず、推奨されるアプローチは先に述べたとおりです。

サーバー側では、Web サーバーまたは Web サーバー拡張機能を Delphi で作成するのに複数のアーキテクチャを使用できます。スタンドアロン Web サーバーの場合は `IdHttpServer` コンポーネントを使用できるのに対して、Web サーバー拡張機能 (CGI アプリケーション、ISAPI、Apache モジュール) を作成する場合は `WebBroker` フレームワークを使用できます。Delphi 2010 では、`DataSnap` の HTTP サポートにより、また新しい選択肢が用意されています。

### XML

XML (Extended Markup Language) がよく使用されるデータ形式ですが、多くの REST サーバーでは XML の代わりに JSON (JavaScript Object Notation) などのデータ構造も使用され、時には、コンマ区切りのテキスト ファイルさえ使用されます。やはり、XML も非常に普及しているので、ここでは詳しく扱いません。

Delphi では、`XmlDocument` コンポーネントを使って XML ドキュメントを処理できます。これは、利用可能な XML DOM エンジンのいずれか（デフォルトは Microsoft XML DOM）に対するラッパーです。いったんドキュメントがロードされれば、XPath を使ってそのノード構造をナビゲートしたりドキュメントについてのクエリを実行することができます（これは私が好んで使うことが多いスタイルです）。

## XPATH

XPath は、XML ドキュメントのノードの特定と処理が可能なクエリ言語です。XPath で用いられる表記法は、ファイル システム パス (/root/node1/node2) に似ていますが、ノード属性やサブノードについての条件 (root/node1[@val=5]) や複雑な表現を表すための角かっこが追加されています。XPath 文の結果は、ルールに一致するノードの数やノード セットの合計値などのように、それ自体が表現になり得ます。

Delphi では、特定の DOM がサポートしている場合は、ドキュメントのホストとなる DOM に要求することで XPath 要求を実行できます。以下の最初のデモでは XPath の例を見てみましょう。

## DELPHI で作成された REST クライアント

Web 上で見つかる REST サーバーの例は、古典的な Amazon Web サービス ("Amazon Web サービス" という名称がクラウド コンピューティング サービスに使用されるようになったため、現在は "Amazon E コマース サービス" にブランド名が変更されています) から、HTML 形式ではなく XML データ構造を使って情報にアクセスできる多くのサイトまで、数え切れないほどあります。

REST を使用する Web サービスの数がインターネット上で多いとしても、(一部のデモで示しているように) 実際の大半の Web サービスには何らかの開発者トークンが必要なものに対して、誰でも自由にアクセスできる Web サービスはほんの一握りです。Delphi REST クライアントの少し異なるリストとこれらのデモのソース コードについては、私の以下の Web サイトを参照してください。

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

## RSS フィード用の REST クライアント

XML 形式で情報を配布するための手段として最も広く知られているのは RSS フィードと ATOM フィードで、これらは主にブログ サイトやニュース サイトに付加されていますが、どのようなデータソースにも同じように使用可能です。フィードに関して興味深い点は、ユーザーが Web ブラウザを使って一般にアクセスするのと同じ情報をクライアント アプリケーションに提供する点です。フィード情報はこれらのクライアント アプリケーションで処理され、時には (DelphiFeeds.com サイトで行われているように) 類似フィードの要約にまとめられることもあります。

REST を使ったクライアント アプリケーションの最初の例として、Delphi Blogs (<http://www.delphifeeds.com>) の内容を調べる非常に簡単な RSS クライアントを作成したのはそのためです。URL を使って動的な XML データにアクセスし、その URL を変更してさまざまなデータにアクセスできる場合は常に、REST アプローチを用いています。

## REST 呼び出しと XML ドキュメント

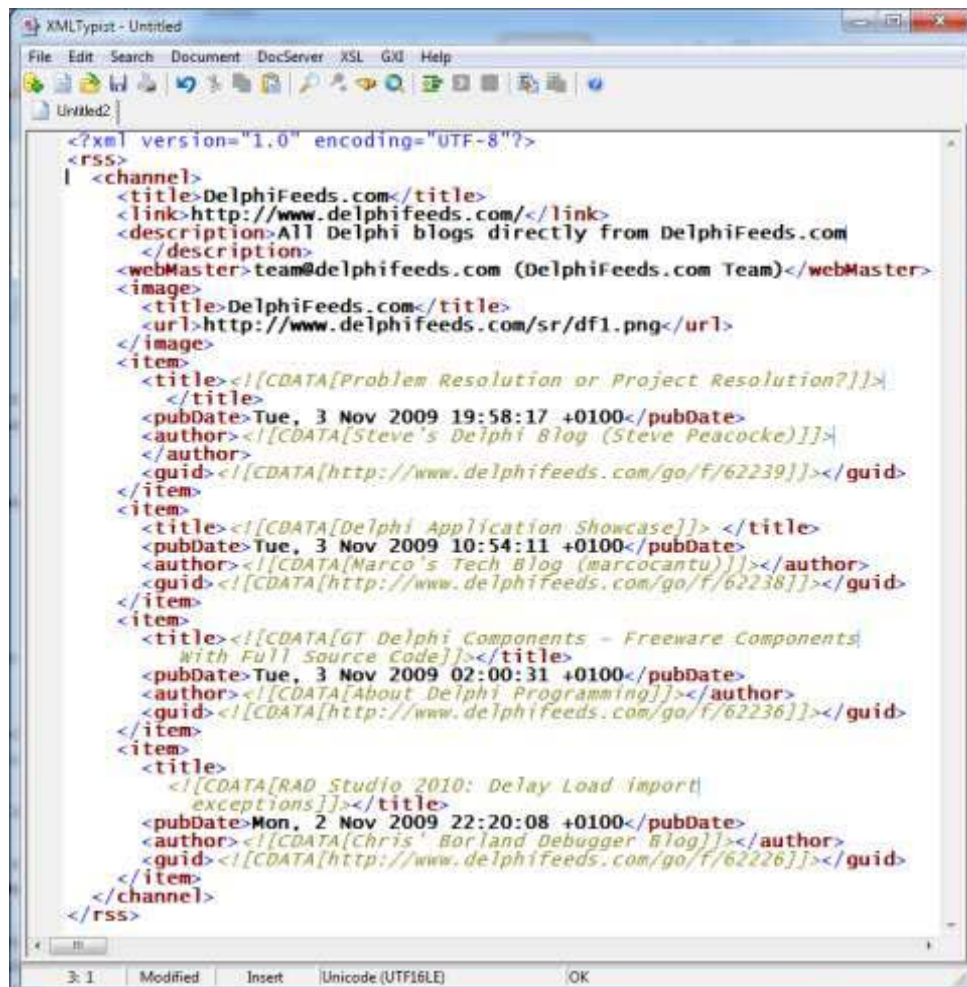
RssClient プログラムでは、IdHttp コンポーネントと XMLDocument コンポーネントを使用します。以下のように、前者は Web からデータを取得する (つまり、REST クライアント呼び出しを行う) のに使用され、後者のコンポーネントにデータをロードします。

```
var
  strXml: string;
begin
```



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

データを抜粋したものを XML エディタに表示すると、以下のようになります（読みやすくするために、やや簡略化してあります）



## XPATH で RSS データを処理する

この XML ドキュメントから関係のある情報を抜き出すために、RssClient プログラムでは XPath 式を使用します。たとえば、リストの先頭のブログ投稿メッセージ（item[1]）のタイトルを読み取るには、以下を使用します。

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

これは他の情報の抽出と一緒に 1 つのサイクルで行われ、データが書式設定されてリスト ボックスに表示されます。XPath を使用する場合は、Microsoft エンジンのカスタム インターフェイスを使用する必要があり、したがって拡張インターフェイス IDOMNodeSelect へのキャストが必要です。

プログラムでは、目的とするノードが見つかったら、私がこのために作成したヘルパー関数 `getChildNodes` を使って子テキスト ノードを探し、データをリスト ボックスに追加します。プログラムの [Update] ボタンがクリックされたときに実行されるメソッドの完全なコードは以下のとおりです。

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

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

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

このプログラムを実行したときの効果については、以下のスクリーンショットで確認できます。





## 地図と位置

住所に関係があるアプリケーションは多いので、位置情報や地図情報へのアクセスはさまざまな状況で非常に役に立つことがあります。近年、Google、Yahoo、Microsoft などの複数の大手サイトにより、Web 上で利用できる地図データがますます増えてきました。

### GOOGLE ジオコーディング サービス

このカテゴリのサービスで私が最初に取り上げるのは、Google のジオコーディング サービスです。このサービスでは、次のような要求で住所を送信して、その緯度と経度を取得できます。

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

以下の画像でわかるように、同じような URL をブラウザに入力してテストすることもできます。この画像では、ニューヨークの座標を XML 形式でブラウザに表示しています。



私が作成したサンプル `GeoLocation` では、Delphi に付属している標準的なサンプル データベース `Customer.cds` に格納されている企業の住所を使用します（ZIP ファイル内のローカル コピーのほかにプロジェクト ソース コードも使用します）。

類似した多くのサービスと同様に、これは限定的な利用については無償ですが（プログラムには、毎分の最大利用率に達するのを避けるための特別な `sleep()` 呼び出しがいくつか含まれています）、利用に当たっては <http://code.google.com> で特定のサービスを登録する必要があります。

デモ プログラムでは、ユーザーの `devkey` を `GeoLocation.ini` ファイルに追加する必要があります。このファイルはユーザーのドキュメント フォルダに存在しなければならず、以下のような単純な構造になっています。

```
[googlemap]
devkey=
```

## 顧客の住所を割り出す

プログラムは 2 つのステップで動作します。

- まず、ClientDataSet コンポーネントをスキャンし文字列リストを埋めることで、市、州(県)、国の一意な名前を探します。このコードは REST とは無関係なので、ここでは省略しました。
- 2 番目のステップは、それぞれの都市を Google ジオコーディング サービスで検索し、その結果得られた情報をインメモリの ClientDataSet に入力することです。

今度は、XML 形式のデータを要求するのではなく、もっと単純な CSV 形式を用いました。プログラムでは、StringList オブジェクトを使ってこのデータを解析します。

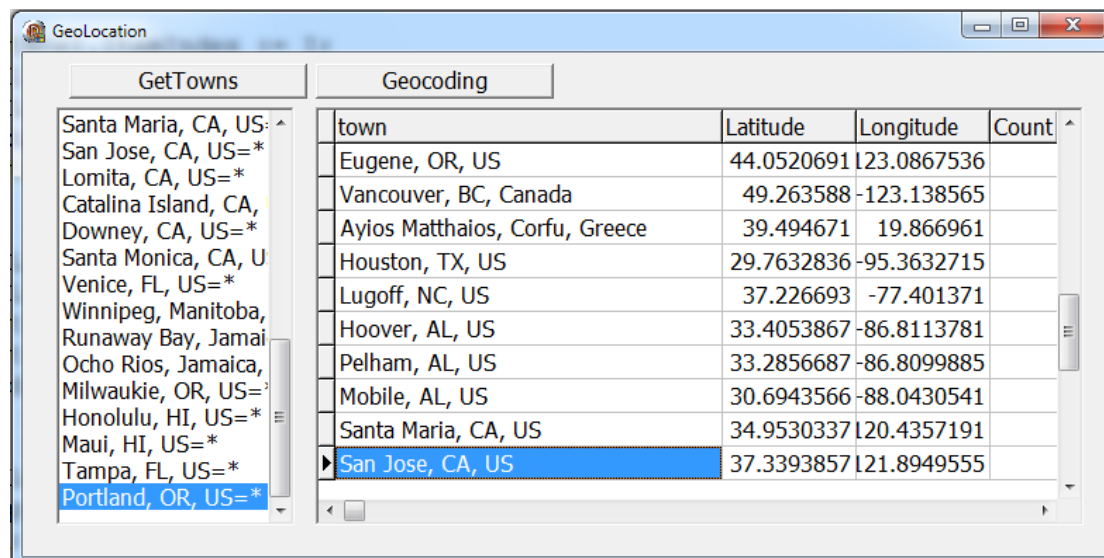
実際のジオコーディング コードは以下のとおりです。

```
procedure TFormMap.btnGeocodingClick(Sender: TObject);
var
  I: Integer;
  strResponse, str1, str2: string;
  sList:TStringList;
begin
  cdsTown.Active := False;
  cdsTown.CreateDataSet;
  cdsTown.Active := True;
  sList := TStringList.Create;

  for I := 0 to sListCity.Count - 1 do
  begin
    ListBox1.ItemIndex := I;
    if Length (sListCity.Names[I]) > 2 then
    begin
      strResponse := IdHTTP1.Get( TIDUri.UrlEncode(
        'http://maps.google.com/maps/geo?q=' +
        (sListCity.Names[I]) + '&output=csv&key=' +
        googleMapKey));
      sList.LineBreak := ',';
      sList.Text := strResponse;
      str1 := sList[2];
      str2 := sList[3];
      cdsTown.AppendRecord([sListCity.Names[I],
        StrToFloat (str1), StrToFloat (str2),
        Length (sListCity.ValueFromIndex[I])]);

      Sleep (150);
      Application.ProcessMessages;
    end;
  end;
  sList.Free;
end;
```

出力は次項の画像のようになるはずですが。



## YAHOO!地図

さらなるステップとして、住所に対応する実際の地図へのアクセスを試みることができます。Google マップには数え切れないほどの機能が用意されているとしても、それはクライアント アプリケーションではなく Web サイトをホストとするように作られています（私にはクライアント プログラムにおける Google マップのホスティングの例はあるにはありますが、そのアーキテクチャとコードはかなり複雑で、このホワイト ペーパーには合いません）。

YahooMaps という新しいサンプルでは、Yahoo!地図 API を使用して実際の地図を取得し、それを Image コントロールに表示します。この REST API に関する情報と無償の Yahoo アプリケーション ID を取得するためのリンクは以下で得られます。

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

この場合もやはり、プログラムを実行するには、この ID を取得して、"ユーザー ドキュメント" フォルダに含まれている YahooMaps.ini という特定の INI ファイルに格納する必要があります。

地図は 2 つのステップで取得されます。最初の HTTP 呼び出しで住所を渡し、地図画像の URL を受け取ります。この URL で示されるデータが、2 番目の HTTP 呼び出しを使って取得されます。この場合もやはり、この 2 つのステップを Web ブラウザでシミュレートでき、デバッグ目的にはとても便利です。

プログラムでは、前のサンプルと同じデータベースと中間の StringList を使用する一方、以下のメソッドを使って、ハードコードされた都市（SanJose, California）の地図を表示するのに使用するボタンも含まれています。

```

const
  BaseUrl = 'http://api.local.yahoo.com/MapsService/v1/';

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

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

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

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

```

最初の HTTP Get 要求で実際のクエリを渡し、実際の地図画像の URL を記述する以下のような XML ドキュメント（長い ID は省略されています）が結果として返されます。

```

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

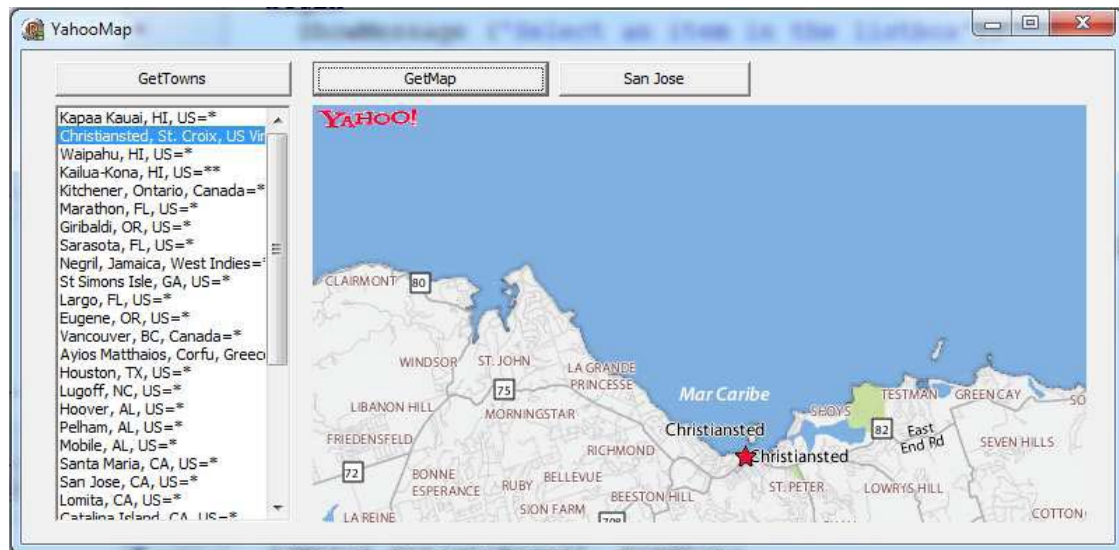
```

プログラムが以下のコードでただ 1 つのノードの値を抽出できるのはこのためです。

```
XMLDocument1.DocumentElement.NodeValue
```

最後に、画像は一時ファイルに保存され、Image コントロールにロードされます。このような特定の都市の地図だけでなく、プログラムでは、前のサンプルで使った Customer.cds データベースに格納されている都市の地図も取得できます。ハードコードされた位置（San Jose：これが Delphi Live デモだったため選ばれました）の地図を取得するためのボタンがあります。

表示画面の例は以下のとおりです。



## GOOGLE 翻訳 API

Google から提供されている REST API の例として、シンプルで興味深いものがもう 1 つあります。それは Google 翻訳 API という翻訳サービスです。ドキュメントは以下に掲載されています。

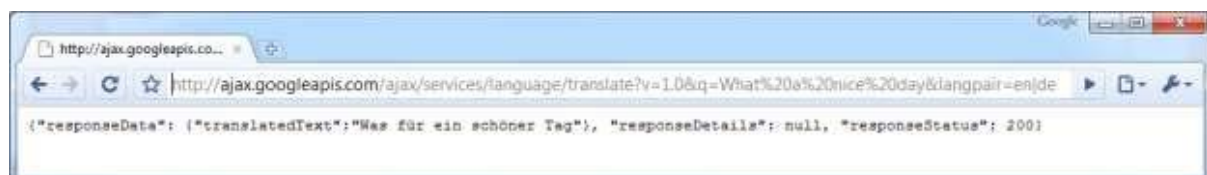
<http://code.google.com/apis/ajaxlanguage/documentation/>

この場合は、サインアップキー（および INI ファイル）は不要で、参照元サイトを指定するだけです（ただし、その情報がなかったとしても万事うまくいくように思われます）。次のような URL を入力することで、Web ブラウザで翻訳を要求することができます。

<http://ajax.googleapis.com/ajax/services/language/translate?>

`v=1.0&q=what%20a%20nice%20day&langpair=en|de`

この呼び出しの出力は以下のようになります（読みやすいように、JSON 形式の結果も示しました）。



```
{
  "responseData":
  {
    "translatedText": "was für ein schöner Tag"
  },
  "responseDetails": null,
  "responseStatus": 200
}
```



このサンプルでは、これまでのデモと比べると、さらにもう一歩踏み込んでいます。HTTP 要求を出すのではなく、クラス メソッドで呼び出される特定のカスタム VCL コンポーネントを使用するので（そのため、可能であったとしてもコンポーネントをフォームに配置する必要はありません）。このサポート コンポーネントにより、API は非常に使いやすくなり、HTTP 呼び出しは完全にカプセル化されます。

## 翻訳コンポーネント

このコンポーネントのクラス宣言は以下のとおりです。

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

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

実際の処理は以下の DoTranslate 関数で行われます。

```
function TBabelGoogleRest.DoTranslate(strIn: string): string;
var
  strUrl, strResult: string;
  nPosA, nPosB: Integer;
begin
  strUrl := Format (
    'http://ajax.googleapis.com/ajax/services/language/translate?' +
    'v=1.0&q=%s&langpair=%s ',
    [TIdUri.ParamsEncode (strIn),
    FFromLang + '%7C' + FToLang]); // format hates the %7 !!!
  strResult := Http1.Get(strUrl);
  nPosA := Pos ('"translatedText":', strResult); // begin of JSON data
  if nPosA = 0 then
    begin
      nPosA := Pos ('"responseDetails":', strResult);
      nPosA := nPosA + Length ('"responseDetails": ');
    end
  else
    nPosA := nPosA + Length ('"translatedText": ');
```

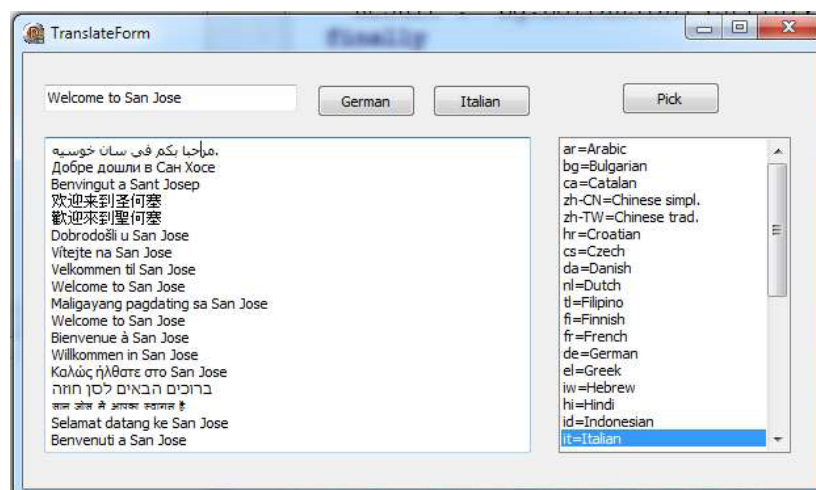
```

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

```

指定された URL への要求の結果は JSON 形式で返されます（これが Google による JavaScript API と見なされているためです）。Delphi 2010 で新たにサポートされるようになった JSON からオブジェクトへのマッピング（後で取り上げます）を用いることもできましたが、ここでは JSON 情報を手動で処理することにしました。実際のメソッドは、一時オブジェクトの生成とそのプロパティの設定および DoTranslate 関数の呼び出しを行うクラス メソッドを使って呼び出すことができます。それ以外のシナリオもありますが、残りのコードはわかりやすいほうがよいでしょう。

デモ プログラムのメイン フォームには、サポートしているすべての言語を一覧表示するリスト ボックスがあります。デモは英語からの翻訳ですが、逆方向（英語への翻訳）にセットアップすることもできます。理論上は任意の 2 つの言語トークンの組み合わせで機能しますが、実際には必ずしもそうとは限りません。いったん翻訳を要求すると、結果がログに追加されます。ここでは、最初の言語グループ（アルファベット順に表示）に呼び出しを適用しました。



## TWITTER — できる限りシンプルに

Twitter Web サービス インターフェイスは、こうしたシンプルなソーシャル Web サイトの出現に大きく貢献しました。これにより、Twitter 対応アプリケーションのエコシステム（生態系）全体が生まれ出されたからです。"Twitter のようにシンプルに" はユーザーに向けたものですが、Twitter の REST Web サービス インターフェイスにも当てはまります。Twitter は特に Ruby で作成されたもので、URL から内部のアプリケーション リソースへの巧みなマッピング（これこそ REST アプローチの基盤です）により Ruby on Rails が REST の考え方を推し進めるのに貢献したことは間違いありません。

Twitter アカунツがある場合、自分の最新エントリ 20 件にアクセスするにはどうすればよいでしょうか。そして、Twitter の最新エントリ 20 件にアクセスするにはどうすればよいでしょうか。その答えは以下の URL です。

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

ユーザー名とパスワード(Delphi で IdHttp コンポーネントの対応するプロパティにたやすく追加できるもの)を含んだ標準の HTTP ヘッダーを渡して GET HTTP 要求を出すことが唯一の条件です。アカウントのステータス更新を投稿することは、いわば、はるかに複雑な操作です。これは、以下のコード断片にあるように、ステータス パラメータを渡す POST HTTP 要求で実現されます。このコードでは、ClientDataSet コンポーネントの現在のレコードのフィールドの 1 つ(ClientDataSet1TEXT)のテキストを Twitter に送信して、それに投稿済みのマークを付けています。

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

このコードは、私の "Delphi Tweet of the Day" アプリケーションから抜き出したもので、つぶやき(ツイート)を入力するのに ClientDataSet (ローカルまたはリモート サーバーに存在)を使用しています。実際には、テーブルの TEXT フィールドの値を投稿したあと、プログラムは POSTED フィールドを 'T' (True) に設定します。

プログラムにはこれ以外のコードもありますが、Twitter そのものまたはその REST API に実際に関係するものではありません。ここでの議論に関係のある要素としては、この他に、以下のような IdHttp コンポーネントの構成があります。

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

言い換えれば、Twitter に対するコーディングはきわめてシンプルで、定期的に、あるいは関係する何か業務に（またはデータベースで）発生したときに自動的に投稿する仕組みは役に立ちます。

## GOOGLE スプレッドシート サービスとの インターフェイス

地図サービスや翻訳サービスを使った作業が非常に快適で、Twitter への投稿が自社のマーケティング活動の一部になり得る場合は、Google ドキュメント（以下を参照）のようなビジネス サービスとやり取りできることは、時間がたつにつれますますます重要になってくると思います。

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

Google ドキュメントの Web サービス インターフェイスでは、Web ドキュメントのアップロードとダウンロード（さらには変換までも）、およびアクセス資格情報の管理が可能です。これは確かにとてもすばらしい機能です。しかし、Web 上の指定ドキュメントの内容を取り扱えるインターフェイスも 1 つあります。それが Google スプレッドシート サービス API です。このホワイト ペーパーの執筆時点では、まだ初期ベータ版の段階にあります。

この API を使用すると、公開/非公開を問わずシステムに存在する任意のスプレッドシートの任意のシートの個々のセルにアクセスできます。Web インターフェイスでこれらのドキュメントを誰が参照しようと、ドキュメントの変化がリアルタイムに表示されます。

この API では（他の多くの Google API と同様に）個人的なドキュメントや保存してあるドキュメントにアクセスできるため、これまで使用してきた他の REST API よりも堅牢なセキュリティ レイヤが必要です。よりシンプルなソリューションと比べて、顕著な違いが以下のように 2 つあります。

- この API では、HTTP ではなく HTTPS を使用します。OpenSSL または別の SSL ライブラリをクライアント アプリケーションにフックし、適切な Indy サポート コードを呼び出す必要があります。
- API では認証要求を別途行う必要があり、その結果、しばらくして有効期限が切れる認可トークンが返されます。（同じ IP アドレスを送信元とする）後続の要求にはすべて、このトークンを渡す必要があります。

特定の問題を掘り下げて考える前に、このインフラストラクチャを見ておきましょう（ところで、この認可管理クラスは別の Google Web サービスであるプロビジョニング API 用に作成したものです。この API は、非公開かつ有料の Google メールおよび Google ドキュメントのドメインでアカウントをセットアップするためのものです）。

認証をサポートしこのサポート機能を使って要求を出すために、TGoogleAuth という専用の Delphi クラスを作成しました。このクラスには以下の public インターフェイスがあります（多くの private フィールドおよびプロパティ アクセス メソッドは割愛しました）。

```
type
  TGoogleAuth = class
  public
    property GmailEmail: string
```

```

    read FGoogleEmail write SetGoogleEmail;
    property GooglePwd: string
        read FGooglePwd write SetGooglePwd;
    property AuthString: string

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

```

上記プロパティのうちの4つ（電子メールアドレス、パスワード、アカウントタイプ、サービス名）は入力値であるのに対して、残りの2つは認可文字列とその設定時刻です。この時刻は、所定の回数だけ自動的に更新するのに使用されます）。このトークンを要求すると、クラスは新しいトークンを取得する必要があるかどうか調べます。AuthString プロパティの取得アクセサメソッドは以下のとおりです。

```

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

```

認可要求コードは Renew メソッドの中にあります。このメソッドでは以下のように、SSL HTTP 接続を使用し、ユーザー名とパスワードを暗号化形式で渡して認可トークンを要求します。

```

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

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

```



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

このクラスのグローバル シングルトンを作成しました。要求を出す必要があるたびに、ヘルパー メソッド（グローバル関数）をパススルーします。このメソッドはトークンをさらに追加します。このかなり長い関数のコードは以下のとおりです。

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

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

```

```

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

```

これはずいぶん長い構造化コードでしたが、SSL ベースの認可済み呼び出しを簡単なものにするだけの価値があります。たとえば、非公開スプレッドシートのリストを要求するためのコードは次のようになります。

```

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

```

このコードは実際のデモの一部です。デモでは、dbtosheet.ini ファイルから取得できる情報を使用します。このファイルは次のような構造になっており、デモ プログラムの実行に必要です。

```

[google]
email=
password=
accounttype=GOOGLE

```

このファイルはプログラムの起動時にロードされ、そこに記述されている 3 つの値が、グローバル シングルトン TGoogleAuth のオブジェクトである googleAuth に値を設定するのに使用されます。4 番目のパラメータであるサービス名は、以下のようにプログラムでセットアップされます。

```

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

```

```
googleAuth.ServiceName := 'wise';
```

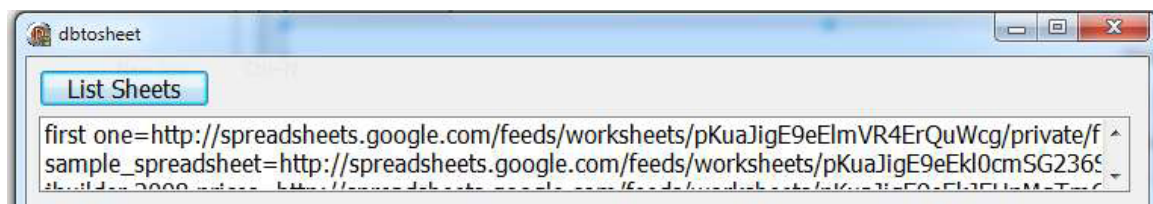
この構成を用意したうえで、プログラムには、利用可能なスプレッドシートのリストを求める上記要求を出してそれらの識別子（または URL）をリスト ボックスに追加するための第 1 ボタンがあります。結果として得られる XML は以下のように XPath を使って解析されますが、これは名前空間を取り除いたあとでの話で、そうでないと、XPath 要求で名前空間を考慮に入れなければならない、非常に複雑なものになります。

```
procedure TForm34.btnListSheetsClick(Sender: TObject);
var
  strXml: string;
  IDomSel: IDOMNodeSelect;
  Node: IDOMNode;
  I, nCount: Integer;
  title, id: string;
begin
  strXml := DoAppRequest ('get',
    'http://spreadsheets.google.com/feeds/spreadsheets/private/full',
    '');
  strXml := StringReplace (strXml,
    '<feed xmlns=' 'http://www.w3.org/2005/Atom' ' ' ' <feed ', []);

  XMLDocument1.LoadFromXML(strXml);
  XMLDocument1.Active := True;

  IDomSel := (XMLDocument1.DocumentElement.DOMNode as IDOMNodeSelect);
  nCount := IDomSel.selectNodes ('/feed/entry').length;
  for I := 1 to nCount do
  begin
    Node := IDomSel.selectNode(
      '/feed/entry[' + IntToStr (i) + ']/title');
    title := getChildNodes (Node);
    Node := IDomSel.selectNode(
      '/feed/entry[' + IntToStr (i) + ']/content/@src');
    id := getChildNodes (Node);
    ListBox1.Items.Add(title + '=' + id);
  end;
end;
```

以下のように、for ループで各ノードを処理し（XPath 関数 count を使用してノード数を決定します）、スプレッドシートのタイトルとその ID/URL をリスト ボックスに追加します。



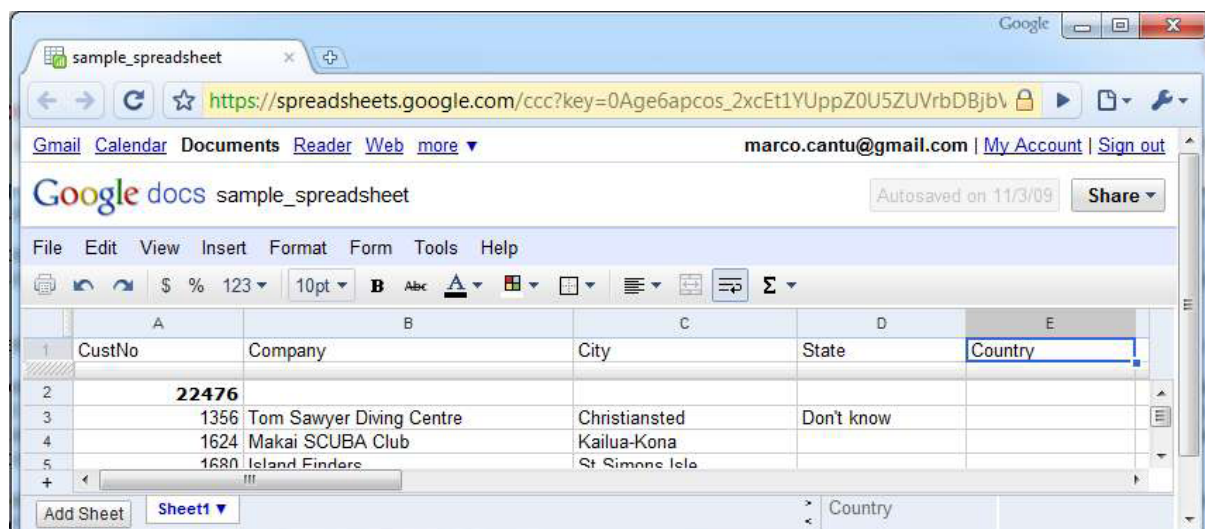
さて、ID/URL が与えられると、2 番目のクエリを発行して、指定ドキュメント内の利用可能なタブシートを要求できます。それが 1 つしかなかったとしても、ドキュメントの内容に影響を及ぼす各操作では特定のタブを参照する必要があります。プログラムでは、以下のようにして、リスト ボックスから ID を取得し 2 番目の REST 要求を出します。

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

この時点で、似たようなループがもう 1 つ登場します。これは、シート名を取り出して 2 番目のリスト ボックスに追加するためのものです。

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

さて、このプログラムが真価を発揮するには、実際のドキュメントにデータを追加できる必要があります。これを実現する最も簡単な方法は、データベース テーブル（すなわち、1 行目にフィールド名があるもの）のように動作するドキュメントをセットアップすることです。以下は、ブラウザ内でこの種のドキュメントを表示したところです。



1 行目のセルは通常のテキスト エントリで、特別なことは何もありません。ここですばらしいのは、以下のように、動的な名前空間を使って、対応する列を参照できる点です。

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

つまり、対応する名前の付いたテーブルのフィールドをフェッチすることで、ドキュメントに新しい行を追加できるということです。それには、シート ID を使った以下のような POST HTTP 要求が必要です。

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

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

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

以下の recordtoxml 関数では、プログラムで使用する ClientDataSet の現在のレコードから、目的とするフィールドの値を取得し、適切な入力データを生成します。

```
function recordtoxml: string;
begin
  Result :=
    FieldToXml ('custno') +
    FieldToXml ('company') +
    FieldToXml ('city') +
    FieldToXml ('state') +
    FieldToXml ('country');
end;

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

既に述べたとおり、gsx: という疑似名前空間はスプレッドシート列の名前を参照し、それらはシートの 1 行目の文字列で決定されます。このコードにより、データベース レコードに対応する新しい行をスプレッドシートに追加することができます。



これで、ドキュメントに対する編集権限のあるユーザーは誰でも、いったん投稿されたデータを編集できるので、ドキュメントはクライアント アプリケーションから送信されたデータに基づいて全体を自動的に再計算でき、その結果得られたドキュメントは、適切な権限のあるユーザーであれば誰でも参照できるようになります。

というわけで、データベース テーブルから得られたデータを表示および編集するための高度なメカニズムができあがりました。データは時間の経過と共に自動的に更新され、堅牢なアクセス制御インフラストラクチャを備えています。それもこれもすべて、無償の Gmail アカウントのおかげです。

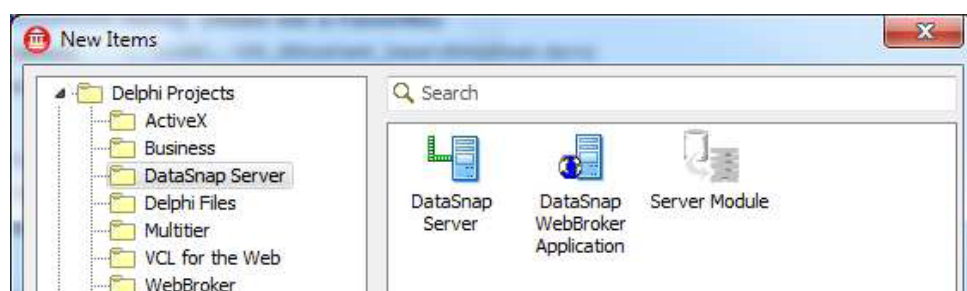
## DELPHI 2010 での REST サーバーの作成

さて、これまでかなりの時間を割いて、Delphi で作成されたさまざまな REST クライアント アプリケーション（さまざまなアクセス許可要求に対処し、さまざまなデータ形式を用いています）を見てきましたので、そろそろ、このホワイト ペーパーの 2 番目のテーマを検討してもよい頃です。後半部では、Delphi 2010 での REST サーバーの作成を重点的に扱います。

誤解しないでいただきたいのですが、Web サーバーを作成できる Delphi（おそらく Delphi 3 以降）であればどのバージョンでも REST サーバーを作成できます。現在では、IdHTTPServer コンポーネントまたは WebBroker アーキテクチャなどのサードパーティ製ソリューションを利用して、独自のカスタム REST サーバーを構築できます。私の著書『Mastering Delphi 2005』に収められている例でおわかりいただけるように、私は長年にわたってこれに取り組んできました。ですので、ここではその選択肢の 1 つに的を絞ってじっくり解説します。それは Delphi 2010 にうまく統合されており、おそらく Embarcadero Technologies から製品に対して今後提供されるあらゆる機能拡張の基礎になると思われるからです。

### 初めてのシンプルな REST サーバーを作成する

Delphi 2010 で初めてのシンプルな REST サーバーを作成する場合は、DataSnap ウィザードを利用できます。

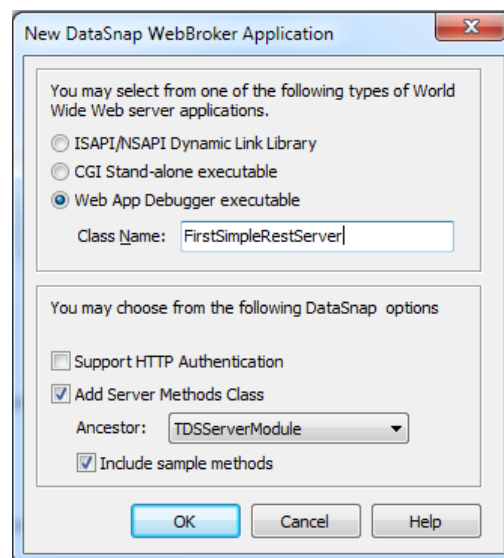


（上記のスクリーンショットでわかるように）実際には 2 種類の DataSnap ウィザードがあるので、どちらを使用するかを決める必要があります。REST サーバーを Web サーバーとしてホストする場合は、[DataSnap WebBroker アプリケーション] を選択するのがおそらく最も良いと思われますが、どちらのウィザードで生成したプロジェクトでも、後からコンポーネントを追加して、それらを非常に似たものにすることができます。一般に、Delphi クライアントなどのクライアント アプリケーションを用意する予定で、サーバーが組織内のネットワークに存在する場合は、通常の DataSnap サーバーのほうが理にかなっているはずですが、オープン アーキテクチャの場合や、ブラウザベースの

アプリケーションから REST サーバーを呼び出す場合は、HTTP サーバーと統合される DataSnap WebBroker サーバーのほうが妥当です。

観点を少し変えると、DataSnap WebBroker サーバーのほうが、着信する HTTP 要求を細かく管理でき、WebBroker サーバー内で REST データを統合することができます。

このオプションを選択すると、以下のような、設定項目がいくつかあるダイアログ ボックスが表示されます。このウィザードで行わなければならない最も重要な決定は、アプリケーション スケルトンのタイプです。ただし、後から実際のコードをすべてそのまま保ちながらプロジェクト アーキテクチャを変更することができるので、あまり気にしないでください。



Web サーバーに実際にホスティングする場合は、ISAPI オプションが一般に推奨されますが、（それより一般に低速の）CGI のほうがセットアップとデバッグが容易です。しかし、デバッグとテストをしっかりと行う場合は、Delphi に用意されている Web アプリケーション デバッガ インフラストラクチャを使用することを本当にお勧めします。このインフラストラクチャでは、コードに直接ブレークポイントを追加できますし、ヘッダーなどの HTTP データ フローさえ調べることができます。そのオプションを選択する場合は、内部クラス名を指定します。このクラス名は（Web アプリケーション デバッガの URL 内で）プログラムへの参照としてのみ使用されます。なお、ウィザード ダイアログ ボックスでこれが明示的に指定されていなくても、Delphi では Apache サーバー統合モジュールをサポートしますが、この種のプロジェクトは手動でセットアップしなければなりません。

DataSnap WebBroker ウィザード ダイアログの下部では、すぐに使用できるサンプル メソッド付のサーバー メソッド クラスを作成することができますが、この場合も、独自のものをコーディングするのはとても簡単でしょう。これらの設定に従って、Delphi では以下の 3 つのユニットから成るプロジェクトを生成します。

- メイン フォーム（特定の用途はありませんが、情報のログ記録に使用できます）。プロジェクト構造を Web サーバー モジュールか CGI に変更すると、このフォームは削除されるため、これについてはあまり気にしないでください。
- DataSnap サーバー コンポーネントのホストとなる Web データ モジュール（TWebModule から継

- 承します)。
- サーバー クラスの役目を果たすデータ モジュール (REST サーバーに実行させるコードをここに追加します)。

## DATASNAP WEBBROKER ウィザードで生成される WEB データ モジュール

プログラムにコードを追加してテストする前に、これら 2 つのデータ モジュールをもう少し詳しく見てみましょう。Web モジュールは WebBroker アーキテクチャの中核的要素です。そこには複数のアクションを定義できるほか、任意の HTTP 要求の前処理イベントと後処理イベントが用意されています。この Web モジュールでは、以下の DSHTTPWebDispatcher のサンプルにあるような、指定された URL アクションをインターセプトするコンポーネントを追加できます。

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

このコンポーネントでは、URL が 'datasnap' で始まる要求をすべてインターセプトし、DataSnap の HTTP サポートに渡します。'datasnap' で始まり 'rest' パスを示す要求については、処理は組み込みの REST エンジンに任せられます。つまり、'datasnap/rest' パスの付いた要求は REST 要求と見なされるのです。これら 2 つは文字列なので、変更して別の URL を指定することができます。これについては後で取り上げます。とりあえずは、標準の設定で話を進めましょう。

Web データ モジュールの他の 2 つのコンポーネントは、DataSnap の全体的な基盤を提供し、どのクラス (複数の DSServerClass コンポーネントを追加する場合は複数のクラス) が要求に応答するかを示します。デフォルト設定は以下のとおりです。

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

DSServer コンポーネントは手動であろうと自動であろうと起動しさえすればよいのですが、DSServerClass の構成は通常、ターゲット クラスを返すイベント ハンドラで行われます。これのデフォルト コード (ウィザードで生成されます) は以下のとおりですが、ここでは、対応するユニット (Fsrs\_ServerClass) でホストされる補助的なデータ モジュール クラス (TServerMethods1) を返します。

```

procedure TWebModule2.DSServerClass1GetClass(
  DSServerClass: TDSServerClass;
  var PersistentClass: TPersistentClass);
begin
  PersistentClass := FsrServerClass.TServerMethods1;
end;

```

最後に、Web モジュールには、以下のように、他のあらゆるアクションに対するデフォルトの HTTP 応答（必要最小限の HTML を返すだけ）が用意されています。

```

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

```

このアクションは設計時に以下のように構成されます。

```

Actions = <
  item
    Default = True
    Name = 'DefaultHandler'
    PathInfo = '/'
    OnAction = WebModule2DefaultHandlerAction
end>

```

## DATASNAP WEBBROKER ウィザードで生成される サンプル サーバー クラス

DataSnap WebBroker ウィザードで生成される 3 番目のユニットはサンプル サーバー クラスです。つまり、REST を通じてリモートで呼び出されるメソッドを定義するクラスです。これは、先にコードを示した DSServerClass1GetClass イベント ハンドラを通じて DSServerClass コンポーネントに接続する Delphi クラスで、実際のコードの大半が最終的に定義される場所です。

生成されるスケルトン クラスは、ウィザードでサンプル メソッドを希望したことによるもので、非常にシンプルです。コードを以下に示します。

```

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

```

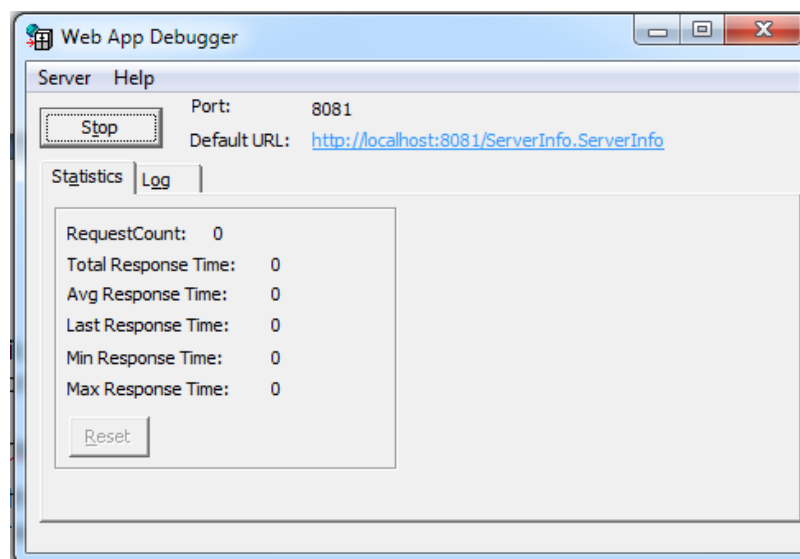
このクラスは TDSServerModule クラスを継承していることに注意してください。TDSServerModule クラスはほとんど標準データ モジュール (DataSetProvide コンポーネントをサポートしています) ですが、public メソッドについての一種の RTTI 生成 (Delphi 2010 の新しい拡張 RTTI より前のもの) を有効にする特別なコンパイラ オプション (`{ $MethodInfo ON }`) を指定してコンパイルされています。

EchoString メソッドはデフォルトでは、渡されたパラメータを返すだけですが、実際の "エコー" のように文字列のトレイルを繰り返すように少し変更しました (以下を参照)。

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

## REST サーバーのコンパイルとテスト

さてここで、サーバーをコンパイルして、うまくいくかどうか確かめることができます。プログラムをコンパイルして実行したあと、Web アプリケーション デバッガ (Delphi の [ツール | Web アプリケーション デバッガ] メニューから利用できます) を実行し、(対応するボタンを使って) 以下のように起動する必要があります。



Web アプリケーション デバッガは特定のポート (私の構成では上記のスクリーンショットでわかるように 8081) で動作します。デフォルト URL を開いて、このアーキテクチャで使用可能なさまざまなアプリケーションを確認するか、目的とするプログラムの特定の URL を "ApplicationName.ServerClass" という形式で入力することができます。

今回は 2 つのトークンが同一なので、サーバーの URL は次のようになります。

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



この URL を Web ブラウザで開くと、Web アプリケーション デバッガと特定のサーバーが動作しているかどうかを確認できます（特定のサーバーは Web アプリケーション デバッガで自動的に起動されないので、既に動作している必要があることに注意してください）。次のような画面が表示されるはずです。

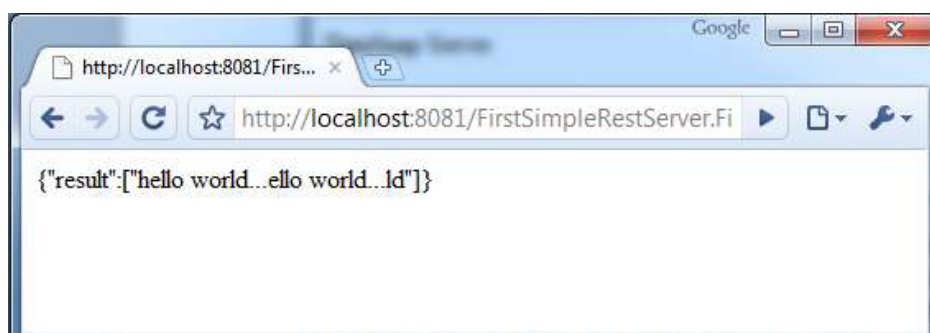


覚えておいてでしょうか、これは標準アクションに対してプログラムから返された HTML です。もちろん、ものすごく興味深いものではありませんが。

次のステップは、REST サーバーで実行できる唯一の要求に対応する特定の URL を使用することです（実行に当たっては、'datasnap' サーバーの 'rest' サポートに関する TServerMethods1 の EchoString メソッドを呼び出します）。この URL は、以下のように REST サーバー プレフィックス（デフォルトは /datasnap/rest）、クラス名、メソッド名、メソッド パラメータを結合することで自動的に生成されます。

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

URL 内の %20 は単にスペースの代わりに使用されているものですが、ブラウザには実際にスペースを入力することができます。以下のように、REST サーバーの JSON 形式の応答が表示されます。



このテストを行っている間も、実際に転送される HTTP 要求と HTTP 応答を把握するために Web アプリケーション デバッガを使用することに注意してください。上記のページは、以下のブラウザ要求によってもたらされたものです。

```
GET /FirstSimpleRestServer.FirstSimpleRestServer/  
datasnap/rest/TServerMethods1/EchoString/hello%20world HTTP/1.1  
Host: localhost:8081
```

```

Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US)
  AppleWebKit/532.0 (KHTML, like Gecko) Chrome/3.0.195.27 Safari/532.0
Accept: application/xml,application/xhtml+xml,text/html;q=0.9,
  text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Encoding: gzip,deflate,sdch
Cookie: LastProgID=FirstSimpleRestServer.FirstSimpleRestServer
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3

```

この要求の結果、以下のような完全な HTTP 応答が生成されます。

```

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

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

```

既に述べたように、このような低レベルの情報にたやすくアクセスできることは、HTTP アプリケーションをデバッグする際には、たいへん幸運なことです。

## 通常の DELPHI クライアントからの REST サーバーの呼び出し

これで、サーバーを作成しそれが動作することを確認したので、それをテストする Delphi クライアント アプリケーションを作成できます。ここで、2 つの異なるアプローチを用いることができます。1 つは、始めに戻り、REST から提供される特定のトランスポート層を使って Delphi DataSnap クライアントを作成することです。しかし、これは、DataSnap の HTTP または TCP トランスポート層を使用する場合と比べて、大きな違いはありません。

もう 1 つの選択肢は、このホワイト ペーパーの前半部で作成したさまざまなクライアント全部とちょうど同じように、カスタム REST クライアントを作成することです（私はこちらのアプローチに従うつもりです）。つまり、特定のサポート機能に頼らないので、クライアント アプリケーションを作成するのに他のどのような言語でも使用できるということです。これを実現するには、標準的な Delphi VCL アプリケーションを作成し、それに、実際の REST 要求を実行する IdHTTP コンポーネント、入力用の編集ボックス、そして以下のコードが定義されたボタンを追加するだけです。

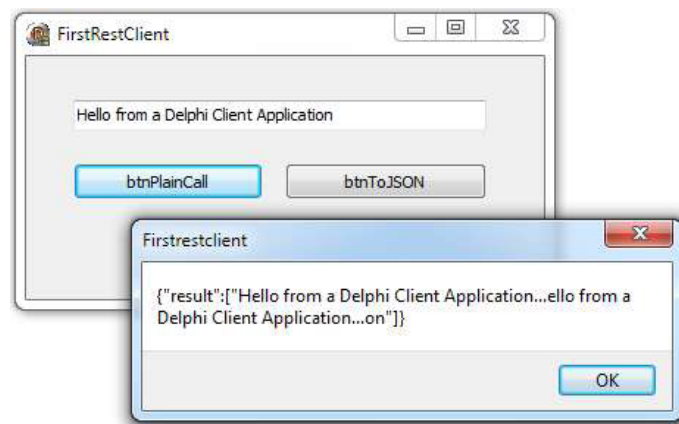
```

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

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

```

この呼び出しでは、サーバーのアドレス、REST サーバーの指定メソッドに到達するための相対パス、そして唯一のパラメータを連結することで、適切な URL を作成します。呼び出しの結果、次の出力が表示されます。



さて、もっと興味深いのは、サーバーから返された JSON データ構造体から実際の情報を取り出すことです。ここでは、前のデモと同じように手動アプローチを用いますが、今回は、Delphi 2010 で利用可能な JSON サポート機能を活用したいと思います。

Delphi の新しい JSON サポート機能は、DBXJSON に定義されている一連のクラスを通じて利用可能です (DBXJSON は、名前に反して、dbExpress フレームワークとは関係のないアプリケーションでも使用できます)。DBXJSON ユニットには、さまざまな JSON データ型 (さまざまな型の個々の値、配列、オブジェクトなど) の操作に使用できるクラスが定義されています。これは、(次のプロジェクトのように) サーバー側アプリケーションの結果をカスタマイズする場合と、(この例のように) クライアントで受け取ったデータを読み出す場合にとっても役立ちます。

サーバーから返される JSON データは文字列ですが、REST サーバー サポート機能では、名前付きの値 (つまり名前と値の "ペア") を持つオブジェクトを作成し、実際の値を配列に格納します。HTTP の結果を解析して JSON データ構造体に格納したあとで、オブジェクトからその中のペアへ、そしてペアからその中の単一要素配列へとたどる必要があるのは、そのためです。

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

```

    ShowMessage ( 'Error in parsing ' + strHttpResult);
    Exit;
end;

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

```

この場合もやはり、サーバーから返されるデータ構造に起因する複雑さがあります。他の状況であれば、結果として得られる JSON の解析とその中の情報へのアクセスははるかに容易でしょう。

## AJAX WEB アプリケーションからの REST サーバーの呼び出し

サーバー側 Delphi アプリケーション間でオブジェクトを受け渡すだけでよければ、JSON を使用する以外の方法もたくさんあります。このやり方は、ブラウザで動作する JavaScript アプリケーションから、Delphi でコンパイルされたサーバーを呼び出す場合に意味があります。Ajax (Web ブラウザ内で行われる非同期 JavaScript 呼び出し) は今も昔も REST の導入を支える推進力の 1 つとなっているため、このようなケースは非常に重要です。ブラウザベースのプログラムから対応する SOAP サーバーを呼び出そうとすると、信じられないほど複雑になります。

それでは、Web ブラウザ内で先ほどのクライアントと同じように動作するアプリケーションを作成するにはどうすればよいのでしょうか。さまざまなアプローチとライブラリを用いることができましたが、今回は、信じられないほど強力なオープン ソース JavaScript ライブラリである jQuery (以下で入手可能) を使用することにします。

<http://jquery.com>

jQuery とその使用法についてここで詳しく述べる時間はないので、せめて、このサンプルの裏で動作する jQuery コードだけでも説明するようにします。まず第 1 に、HTML ページには、以下のよう

```

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

```

第 2 に、ページのユーザー インターフェイスは非常にシンプルで、以下のように、何かのテキスト、入力フィールド、ボタンで構成されます（シンプルさを重視したいので、高度な CSS もグラフィックスの追加ありません）。

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

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

  <p>Insert the text to "Echo":

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

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

これがスケルトンであれば、実際の JavaScript コードはどのようなものなのか見てみましょう。必要な作業は、ボタンへのイベント ハンドラの追加、入力テキストの読み取り、REST 呼び出しの発行、そして最後に結果の表示です。ページ オブジェクトにアクセスするために、以下のように、オブジェクト ID に基づいて、jQuery セレクタの中でも最もシンプルなものを使用します。

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

これは、入力テキストの DOM 要素をラップする jQuery オブジェクトを返します。イベント ハンドラを定義するために、ボタンの click() 関数に無名のメソッド パラメータを渡すことができます。呼び出しはもう 2 つありますが、それは、REST 呼び出しそのもの（グローバル メソッド getJSON を使用）と、出力要素の HTML に結果を追加するための html() 呼び出しです。

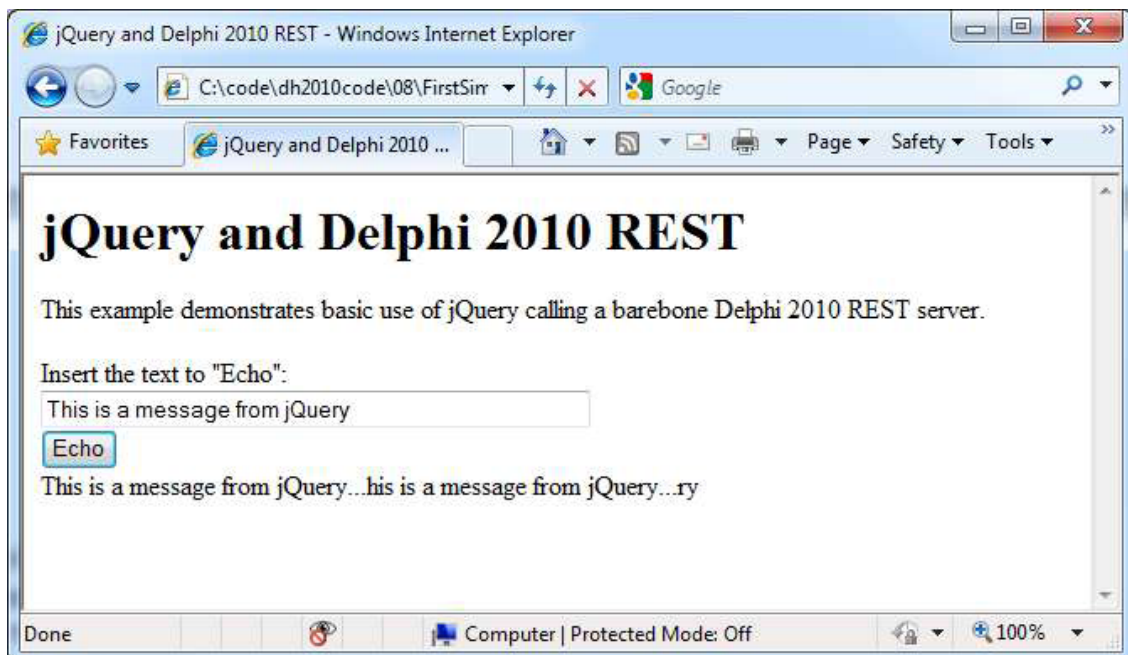
以下はこのデモの中核を成す完全なコードで、非常にコンパクトですが、必ずしも読みやすい JavaScript スニペットです。

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

所定のコードを含んだ HTML ファイルを開くだけで、カスタム サーバーを呼び出すことができますが、それは、ローカル ファイルからローカル REST サーバーへの Ajax 呼び出しがブラウザのアクセ

ス権の設定で許されている場合に限りです。一般に、大半のブラウザでは、HTML ページの生成元と同じサイトの REST サーバーを呼び出せるだけです。

いずれにせよ、Internet Explorer では、ローカル スクリプトを有効にし、限定的なセキュリティ（ファイルがローカル マシン上にあるので選択可能です。ステータス バーのアイコンを参照してください）を希望すれば、このローカル ファイルをうまく処理できるように思われます。



その他のブラウザでは、Web サーバーから HTML ページと REST データが両方とも返されるようにする必要がありますが、今回の REST サーバーは実際に Web サーバーなので、これはそう大変な話ではありません。したがって、“サーバー側” ソリューションの場合に必要な作業は、（Web ブラウザに関する限り）Web サーバー モジュール（“file” URL にフックしたもの）にアクションを追加し、そこから HTML ファイルを返すことだけです。

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

それでは、/file URL で指定のサーバー ページを参照し、JavaScript コードでそのファイルを取得し、そこから REST サーバーを呼び出せるようにすることができます。





これと前の画像との違いは、使用しているブラウザが異なることだけでなく、参照先の URL が異なることです。この 2 番目の例では、ファイルをロードするのではなく、サーバー側 REST アプリケーションを完全な Web サーバーとして使用し、Ajax を通じて同じサーバーを呼び出すための HTML を返します。

## オブジェクトの返却と更新

さてこれで、Delphi 2010 の DataSnap サポート機能による非常にシンプルな REST サーバーの開発について検討したので、そろそろ、サーバーをより強力なものにするためにサーバー上に作成できる実際のコードを解き明かしてみましょう。既に見たように、サーバーは JSON データを返し、関数の結果をこの形式に変換します。結果としてオブジェクトを渡し、変換させることができます。ただし、実際のほとんどの状況では、特定の JSON オブジェクトをサーバー側で作成して返すほうが良いでしょう。それが次のプロジェクトの目標の 1 つになります。

同じプロジェクトで、`get` 以外の HTTP メソッドを処理し、JavaScript で作成されたシンプルなブラウザベース クライアントからサーバー側オブジェクトを取得および変更できるようにする方法も示します。最後に、その過程で、URL 管理に焦点を合わせ、それをより適切で柔軟なものにする方法を解き明かします。

## JSON オブジェクトと JSON 値を返す

この後半のプロジェクトには、DataSnap WebBroker ウィザードを使用し、Web アプリケーションデバッガ アーキテクチャを再び選択しました。そして、REST 呼び出しのターゲット クラスとしてデータ モジュールは必要ないので、基底クラス `TPersistent` を使用することにしました。特定の RTTI サポート機能が必要なので、以下の生成コードにあるように、（少なくとも）`TPersistent` から継承し、クラスに `$METHODINFO` 指令を付けるという規則にします。

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

ご覧のとおり、値が完全なオブジェクトのどちらかを返すため、クラスに少し関数を追加しました。後で、クラスにそれら以外のメソッドも追加します。

このアプリケーションの背後にあるデータ構造はカスタム型のオブジェクトのリストです（これは、もっとオブジェクト指向的に作成することもできましたが、例であることからシンプルさを保つことにしました）。

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

オブジェクトはディクショナリに保存されます。ディクショナリは、Delphi 2009 以降 Generics.Collections ユニットに定義されているジェネリックコンテナクラス TObjectDictionary<TKey,TValue> を使って実装されます。このグローバルオブジェクトは、プログラムの開始時にいくつかの定義済みオブジェクトを追加して初期化されます。必ずオブジェクト名がディクショナリキーと同期し未指定時にはランダム値が設定されるように、特定の AddToDictionary 手続きを使用してオブジェクトを追加している点に注意してください。

```
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');
```

このデータ構造が用意できたので、JSON 値を返すのに使用する最初の 2 つのサンプルメソッドに取り組むことができます。最初のメソッドは、以下のように、与えられたオブジェクトの値を返します（関数にパラメータが渡されない場合はデフォルト値を返します）。

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

(以下の 2 行に示すように) パラメータ付きまたはパラメータなしの URL を使用する場合も、

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

結果として得られた以下のような JSON 応答 (特定のオブジェクトかデフォルト オブジェクトに対応する) を取得します。

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

特定の値ではなく完全なオブジェクトを返す場合はどうなるでしょうか。今回の REST サーバーでは、それを自動的に変換する手段がシステムにないので、TObject 値を返せませんが、実際には、新しい JSON マーシャリング サポート機能を利用して、既存のオブジェクトを JSON 形式に変換できます。

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

このアプローチは、Delphi クライアント アプリケーションでオブジェクトを作成し直さなければならない場合に主に役立ちますが、クライアントが別の言語で作成される場合には、あまり便利ではありません。結果として得られる JSON は、以下のように見た目が少しごちゃごちゃしています。

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

それでは、JSON オブジェクトを返す最良の方法は何でしょうか。サポート クラスを使って、サーバー側にオブジェクトを作成することでしょう。これは、MyData 関数で用いた方法です。

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

```

```
TJSONPair.Create ('Value',
  TJSONNumber.Create(md.Value)));
end;
```

ご覧のとおり、私は TJSONObject を作成し、名前と値を表す 2 つのペアまたはプロパティを追加しました。動的な名前（つまり、ペアの名前部分の値）を使用することもできましたが、そうすると、クライアント側でデータを取得しにくくなります。このコードの結果は、以下のようなすっきりとした JSON コードになるはずです。

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

## TJSONARRAY でオブジェクトのリストを作る

これでオブジェクトのリストができたので、そのオブジェクト リストにアクセスする必要があるでしょう。名前のみのリスト（データがない）にすると、クライアント側のユーザー インターフェイスを作成するときに役に立ちます。

リストを返すには TJSONArray を使用できますが、この場合、これは文字列の配列になります。これらの文字列は、以下のように、ディクショナリのキー（Keys）に対して列挙子を使用して作成するものです。

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

この呼び出しの結果は JSON 形式の配列で、これが今度は（いつものとおり）result という配列に渡されます（したがって、以下のような二重ネスト配列表記になります）。

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

さてこれで、値のリストを返し個々の要素のデータを取り出す手段ができたので、ユーザー インターフェイスの作成に取りかかることができます。

## クライアントを作成する: リストと値

ユーザーに値を選択させるには、値のリストが含まれた初期 HTML を作成するのではなく、Ajax モデルをフルに活用することができます。

起動時のページにはデータはまったくなく、HTML 要素と JavaScript コードしかありません。ページは、ロードされ次第、ユーザーの介入がなくてもサーバーに接続し、実際のデータを要求してユーザー インターフェイスに表示します。

たとえば、起動時にプログラムは、以下の HTML 要素と Ajax 呼び出し（ドキュメントの用意ができたとき、つまり DOM がロードを完了したときに実行されます）を使って、Sample オブジェクトの値を表示します。

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

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

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

MyData への Ajax 呼び出しでは、オブジェクト名を追加の URL パラメータとして渡し、結果の配列から Value というプロパティ/ペアを抽出して、空の HTML span 要素に入れて表示します。リストに対しても、似たようなこと（ただし、もう少し複雑）が行われます。この場合もやはり Ajax 呼び出しがありますが、今度は、結果として返す HTML を作成しなければなりません。この操作は、起動時に自動的にまたはユーザーによって手動で呼び出される個別の *refreshList* 関数（以下のコードを参照）で実行されます。

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

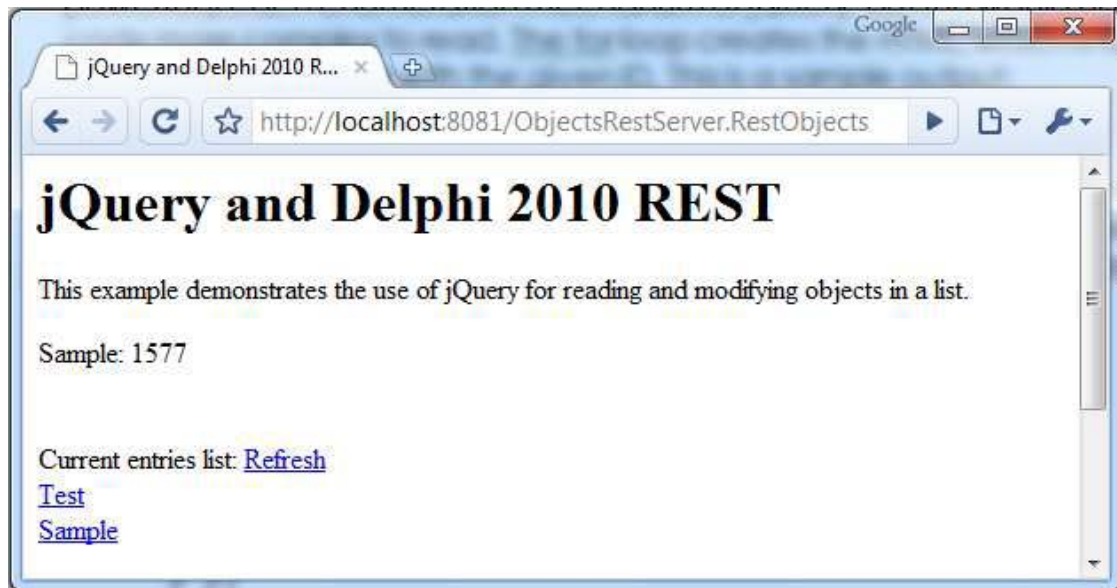
function refreshList()
{
  $.getJSON(baseUrl + "list",
    function(data) {
      thearray = data.result[0];
      var ratingMarkup = ["<br>"];
      for (var i=0; i < thearray.length; i++) {
        ratingMarkup = ratingMarkup +
          "<a href='#'>" + thearray[i] + "</a><br>";
      }
    }
  );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

```

        $("#list").html(ratingMarkup);
    } );
};

```

このコードでは、for ループを使用して、結果の配列をスキャンします。ここで jQuery の `$.each` 列挙メカニズムを用いることもできましたが、そうすると、コードがもっと読みにくくなったでしょう。for ループでは HTML を生成し、それが後ほど指定 ID の span プレースホルダに表示されます。以下はサンプル出力で、Sample オブジェクト（コードは前述のとおり）の値と JSON 配列で返される値のリストが含まれます。



先に述べたように、`refreshList` 関数は起動時に（ready イベント ハンドラで）呼び出されるほか、対応するリンクにも接続されているため、ユーザーは HTML ページ全体を更新しなくても後からリストのデータを更新できます。

```

$(document).ready(function() {

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

```

コード生成については実際にはもう少し説明が必要です。リスト（リンクのリスト）の HTML ができ次第、それらのリンクにコードをフックする必要があります。そうすることで、ユーザーがリストの各エントリを選択したときに、対応するサーバー側オブジェクトをクライアント アプリケーションがロードできるようになります。オブジェクト データのユーザー インターフェイスは 2 つの入力ボックスで構成されます。これらは後でオブジェクト データの操作にも使用されます。この動作は `list` コンテナ内の各アンカーに追加されます。

```

$("#list").find("a").click(function(e) {
    var wasclicked = $(this);
    $.getJSON(baseUrl + "MyData/" + $(this).html(),

```



```
function(data) {
    strResult = data.result[0].value;
    $("#inputName").val(wasClicked.html());
    $("#inputValue").val(strResult);
} );
});
```

\$(this) 呼び出しの使い方に注意してください。これは事実上、Delphi イベントの Sender パラメータになっています。クリックされた要素の html コンテンツはその要素のテキスト（URL に指定してサーバーに渡さなければならない要素の名前）で、以下のような表現になります。

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

これでこのコードができあがったので、リストの要素の 1 つをクリックしたときの効果を確認できます。さらなる Ajax 呼び出しでサーバーに接続して指定の値を要求し、返された値が以下のように 2 つの入力テキスト ボックスに表示されます。



ご覧のとおり、プログラムでは値を取得できますが、最も一般的な操作（いわゆる CRUD インターフェイス：作成、読み取り、更新、削除）を実行するための 3 つのボタンもあります。これは、4 つの HTTP メソッド（PUT、GET、POST、DELETE）を使用する HTML でサポートされています。これらが Delphi 2010 の REST サーバーでどうサポートされているかが次のセクションのテーマです。

## POST、PUT、DELETE

これまで、REST サーバーからデータを取得する方法だけを見てきましたが、データの更新についてはどうでしょうか。REST で広く受け入れられている考え方は、操作を特定するために具体的な URL を使用することを避け、サーバー側オブジェクトだけを特定する URL（たとえば、ここで登場した MyData/Sample など）を使用し、必要な操作を HTTP メソッドで指示するということです。

さて、Delphi の REST サポート機能が URL をメソッドにマッピングするだけのものであったら、うまくいかなかったところですが、実際にはそうではなく、かなりシンプルな仕組みを用いて URL と HTTP メソッドをメソッドにマップします。つまり、以下のようなマッピングを用いて、操作の名前がメソッド名の先頭に付加されます。

- GET get(デフォルト、省略可能)
- POST update
- PUT accept
- DELETE cancel

DSHTTPWebDispatcher コンポーネントの対応する 4 つのイベント ハンドラを操作することで、これらのマッピングをカスタマイズすることができます。さまざまな操作をサポートするために標準の命名規則に従うことにした場合は、サーバー クラスを以下のように定義する必要があります。

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

要素を取得または削除する場合は名前だけが必要なのに対して、作成または更新する場合は、データに関する第 2 パラメータが必要になります。3 つの新しいメソッドの実装はかなりシンプルかつ直接的ですが、値を返す必要がないこともその理由の 1 つです（言うまでもなく、パラメータが空でないことと、サーバー側オブジェクトが実際に存在することは確認すべきでした）。

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

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

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

## クライアント側での編集

REST サーバーでは CRUD 操作を使用できるようになっているので、3 つの編集ボタン（ブラウザ ユーザー インターフェイスの画像は既に示したとおり）のコードを書けば、JavaScript クライアント アプリケーションを完成させることができます。

jQuery には get 操作の固有のサポート機能（先に使用した JSON 固有のものなど、さまざまなバージョンがあります）と post 操作のある程度のサポート機能が用意されていますが、それ以外の HTTP メソッドについては、低レベルで若干複雑な \$.ajax 呼び出しを使用する必要があります。この呼び出しには、12 個を超える可能なペア値のリストがパラメータとして含まれています。もっと重要なパラメータは type と url ですが、データには POST パラメータをさらに渡すことができます。

更新はかなりシンプルで、URL を使ってすべてのデータを REST サーバーに提供できます。

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

削除も同様にシンプルで、削除するオブジェクトへの参照を含んだ URL を作成する必要があります。

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

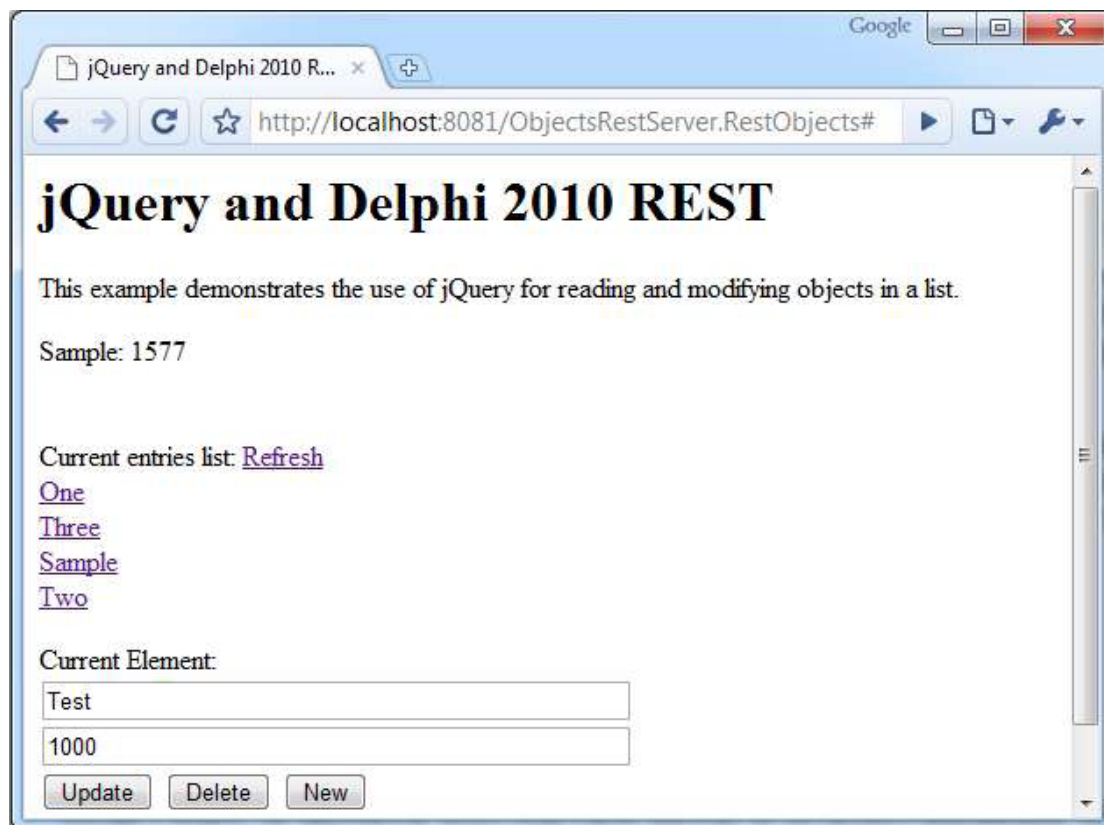
PUT の実装方法を理解するには、もう少し時間がかかりました。と言うのも、データを指定しない場合、一部のブラウザ（特に Chrome）はデータを “undefined” としてポストし、その結果、REST サーバーの HTTP 入力解析がクラッシュしてエラーになるからです。情報を渡す必要がある（しかも、サーバーが必要とする以上のパラメータを渡すことができず、その場合も同様にエラーになる）ので、できることと言えば、URL 要素の 1 つを、対応するデータ要素に置き換えることです。

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

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

なお、jQuery のドキュメントでは、入り交じった結果が得られるおそれがあるため、ブラウザで PUT を使用しないように特に警告しています。また、多数の REST サービス（Microsoft 提供のものも含む）でサーバー側オブジェクトの更新にも作成にも POST を使用する傾向があるのも、それが理由かもしれません。私としては、明快さと一貫性を保つために、これら 2 つの概念はできれば切り離して考えたいと思っています。

これで、3 つの追加メソッドをクラスと適切な JavaScript 呼び出しに追加したので、REST サーバー内のオブジェクトを作成および編集するための完全なブラウザベース ユーザー インターフェイスを備えたサンプルができあがりました。以下の図は、サンプルでオブジェクトをいくつか作成したところです。



## データ指向 REST サーバー

DataSnap の背後にある本来のアイデアが、中間層サーバーからクライアント アプリケーションへのデータ テーブルの移動に的を絞ったものであるのなら、Delphi 2010 で作成された REST サーバーからデータセットを返せないのは、最初はとても奇妙なことに思われるでしょう。そうですね、直接には返せません。言い換えれば、XML 表現を返す場合ほど容易には返せません。しかし、Delphi データセットの全データを格納した JSON 形式の結果を作成することができるのです。それが私の最後のサンプルの主眼です。

プログラムは必要最小限のものだけです。機能が Dataset の全データ（メタデータを除く）を返すことだけだからです。機能拡張できる部分はいくつかありますし、洗練されたユーザー インターフェイスもありますが、皆さんの出発点にはなるはずです。サーバー クラスにはメソッドは 1 つしかなく、データセット全体（個々のオブジェクトやレコード）を JSON 配列に格納して返すだけです。

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

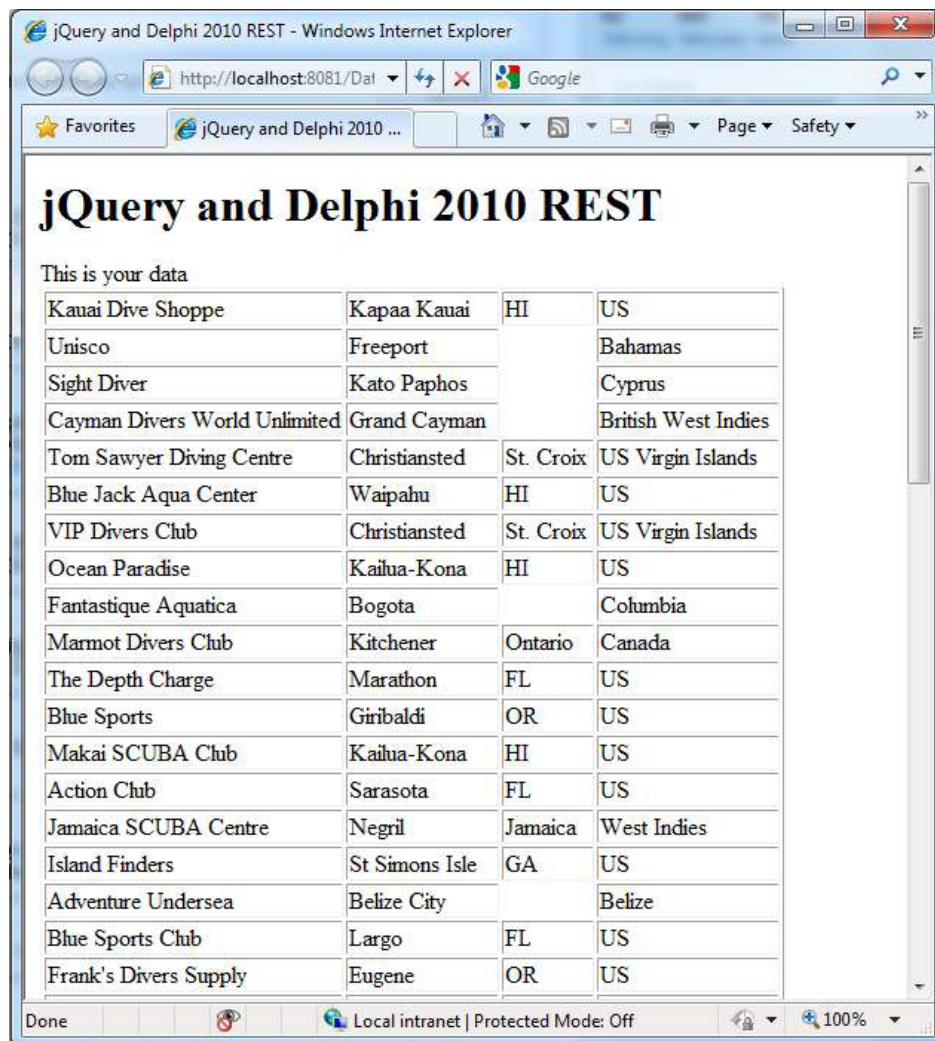
  while not ClientDataSet1.EOF do
  begin
    jRecord := TJSONObject.Create;
    for I := 0 to ClientDataSet1.FieldCount - 1 do
      jRecord.AddPair(
        ClientDataSet1.Fields[I].FieldName,
        TJSONString.Create (ClientDataSet1.Fields[I].AsString));
    Result.AddElement(jRecord);
    ClientDataSet1.Next;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

このメソッドは、ページのロード後にクライアント アプリケーションから呼び出され、以下の jQuery コード（今ではもう、このコーディング スタイルはおなじみのものになったはずです）で HTML テーブルを動的に作成します。

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

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

以下のように、結果を Internet Explorer で確認できます。



もう少し改良できるでしょうか。表示するフィールドがすべて必要な場合は、それらをすべて JavaScript で一覧表示しないようにするというのはどうでしょうか。プログラムの最終版では、何らかのメタデータ サポートを追加して、最終的な出力を改善します。

サーバー側には、データセットのフィールド定義から得られるフィールド名の配列を返す 2 番目のメソッドがあります（以下を参照）。

```
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;
```



クライアント側 JavaScript は拡張され、メタデータを取得するための呼び出しが REST サーバーへの 2 番目の呼び出しとして追加されました（以下を参照）。

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

この情報は、テーブル ヘッダーの作成とオブジェクト プロパティへの動的アクセス（object["propertyname"] という表記で行われる）に使用されます。プロパティ シンボルでオブジェクトにアクセスするのに使用した既存コードは以下のとおりですが、

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

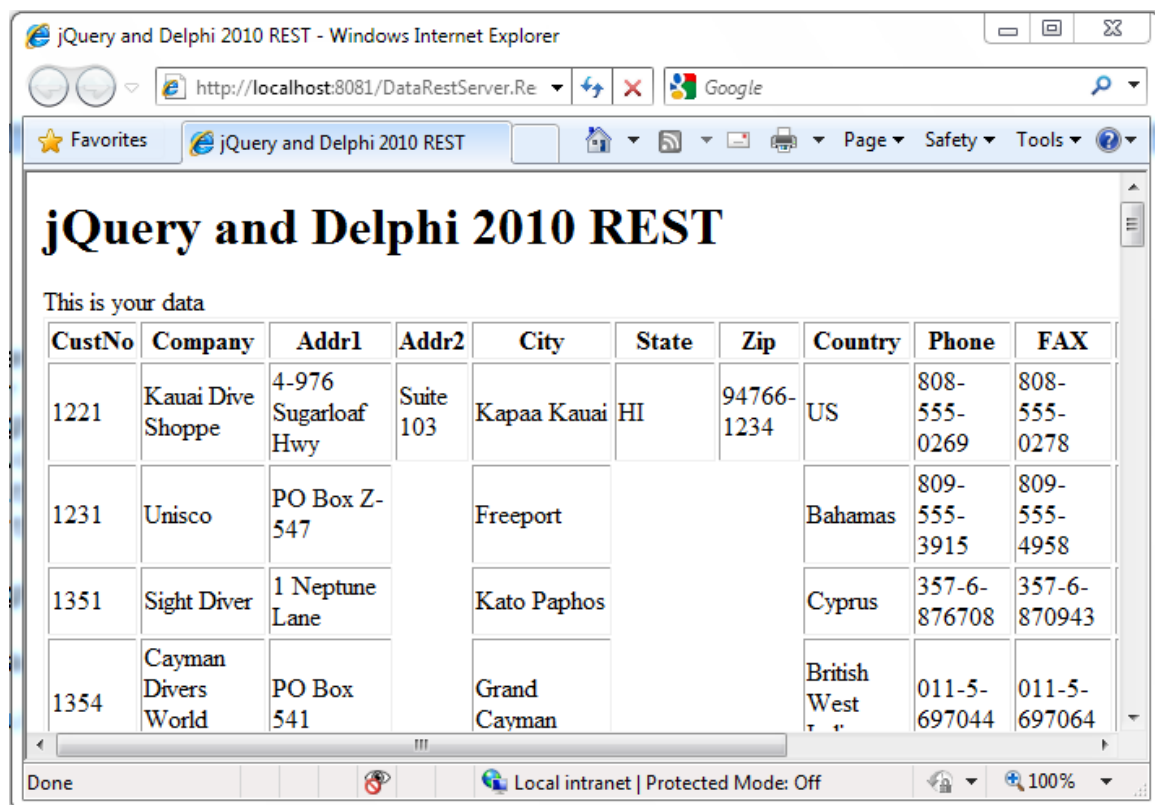
これは、メタデータに格納されているフィールドの名前を使って名前でプロパティを読み取る以下のようなコードになりました。

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

HTML マークアップの作成に使用する完全な JavaScript コードは以下のとおりです。

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

この拡張版の出力は少し見栄えが良くなりました（柔軟性も高まりました）。



CustNo	Company	Addr1	Addr2	City	State	Zip	Country	Phone	FAX
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI	94766-1234	US	808-555-0269	808-555-0278
1231	Unisco	PO Box Z-547		Freeport			Bahamas	809-555-3915	809-555-4958
1351	Sight Diver	1 Neptune Lane		Kato Paphos			Cyprus	357-6-876708	357-6-870943
1354	Cayman Divers World	PO Box 541		Grand Cayman			British West	011-5-697044	011-5-697064

繰り返しになりますが、このプログラムは出発点となることを意図したもので、jQuery プラグインは何も使用していません。このプラグインを利用すれば、HTML テーブルは機能が大幅に増え、ソート、フィルタリング、編集などの機能を備えた強力なユーザー インターフェイス グリッドに変わでしょう。

## まとめ

このホワイト ペーパーでは、REST アーキテクチャと、Delphi 2010 による REST クライアント/サーバー アプリケーションの開発を少しかじってみました。関係する技術（XML、JSON、XPath、JavaScript、jQuery）についてはごく限られた概要だけを説明しました。REST アーキテクチャを使いこなせるようになるには、これらの技術の詳細を習得する必要があるでしょう。

公開される REST サーバーの数が増え、クラウド コンピューティングが出現し、Web にホストされるアプリケーションへの関心が高まるにつれて、Delphi は、リモート サーバーを呼び出すリッチ ユーザー インターフェイス クライアントの開発においても、（任意の言語で作成された）クライアント アプリケーションまたは直接ブラウザでデータ構造を操作するための実際のサーバーの開発においても、重要な役割を果たすことができます。

最後のデモで示したように、JavaScript と Delphi REST サーバーを組み合わせることで、高品質で近代的なプロフェッショナル向け Web ベース アプリケーションの開発に Embarcadero IDE を使用できるようになります。

## 著者略歴

Marco Cantù 氏は、ベストセラーになっている『Mastering Delphi』シリーズ本の著者で、近年は、『Delphi 2007 Handbook』、『Delphi 2009 Handbook』、『Delphi 2010 Handbook』（まもなく完成します）など、Delphi の最新版に関する書籍を自費出版しています。

Delphi についてのトレーニングとコンサルティングのほか、同氏は、他のサーバーの呼び出しと世の中への公開という観点から、Web アーキテクチャおよび Delphi プロジェクトと Web の統合に関するコンサルティングも行っています。

同氏のブログは <http://blog.marcocantu.com>、Twitter は <http://twitter.com/marcocantu> でご覧いただけます。また、連絡先は [marco.cantu@gmail.com](mailto:marco.cantu@gmail.com) です。

このホワイト ペーパー（特に Delphi 2010 の JSON サポート機能に関する部分）の執筆にご助力いただいた Daniele Teti 氏に感謝いたします。



## エンバカデロ・テクノロジーズについて

エンバカデロ・テクノロジーズは、アプリケーション開発者とデータベース技術者が、多様な環境でソフトウェア・アプリケーションを設計、構築、実行するためのツールを提供しています。米国企業の総収入ランキング「フォーチュン 100」のうち 90 以上の企業と、世界で 300 万以上のコミュニティが、エンバカデロの CodeGear 製品や DatabaseGear 製品を採用し、生産性の向上を実現し、オープンコラボレーションとイノベーションを可能にしています。設立は 1993 年、サンフランシスコに本社を置き、世界各国に支社を展開しています。詳細は、[www.embarcadero.com/jp](http://www.embarcadero.com/jp) をご覧ください。