White Paper

embarcadero

# Mobilizing your Business with Enterprise Mobility Services Middleware

By Cary Jensen

April 2015

# Table of Contents

# Executive Summary

The Enterprise Mobility Services feature of RAD Studio is a set of services that provide a turnkey solution for building multi-tier applications. Enterprise Mobility Services, or EMS, consists of two services. The first is a REST (REpresentation State Transfer) web service that supplies a ready-made administrative API (application programming interface), and which can easily be extended with custom endpoints. The administrative API provides REST endpoints that support the creation and management of users and groups. Custom REST endpoints permit you to selectively expose the features and data required by your client applications.

Because EMS uses REST, EMS features can be accessed from a wide range of client applications, from traditional desktop applications to mobile apps, from web pages using JavaScript to third party web services. Mobile clients, in particular, benefit from the ease with which data can be accessed from your EMS server, since these platforms typically lack the ability to connect to remote databases directly. And because all EMS endpoints can enforce authentication and authorization, you can control who can access your data and from where.

The second service provided by EMS supplies you with analytics that give you detailed information about which users and user groups are accessing your EMS service, and how. This service tracks user access by the hour, day, month, and year. It also reports which of your REST endpoints are called and when. This information can help you make decisions about which areas of your applications deserve your development resources, and which users are taking advantage of your services.

In addition to the two services already mentioned, your EMS license includes a license for Embarcadero's InterBase database server for secure enterprise SQL data storage that can be used by your EMS service, and InterBase ToGo licenses, which permit you to install secure, transaction-based databases on your mobile EMS clients.

# Introduction

Enterprise Mobility Services (EMS) is a multi-tier framework that first shipped with RAD Studio XE7. This white paper is designed to provide you, the software professional, with the information you need to get started with EMS today.

This paper begins with a general overview of EMS. In this section you will learn how EMS interfaces with the various technologies necessary to support your applications, and why this approach is so valuable in today's distributed computing landscape.

The paper continues by showing you how to get started with your EMS development. This section begins by showing you how to install EMS and make basic calls into this service. Importantly, these calls are standard REST (REpresentational State Transfer) calls, which can be emitted by any language or framework that understands HTTP (HyperText Transfer Protocol). After a brief discussion of REST, you will learn how to make some of these basic calls using a browser, RAD Studio's REST Debugger, and a Delphi application.

Next, you will learn how to extend the provided administrative API (application programming interface) with your own custom REST endpoints, making the data and functionality of your service available to the broadest range of client applications. Here you will learn how to create EMS modules, define and implement EMS resources, and access these custom REST endpoints from EMS clients.

With the EMS server now sporting more than its administrative API, it's time to consider security. In this section you will learn how to require a user to identify themselves before they can access EMS resources, a process called authentication. In addition, you will learn how to enable authorization, a mechanism that can restrict particular resources and endpoints to select groups of users.

Up to this point, you will be working with the EMS development server, an HTTP server that ships with almost all versions of RAD Studio (The EMS developer edition does not ship with the Starter Editions of Delphi or C++Builder, and is available as an optional add-on to RAD Studio, Delphi and C++Builder Professional editions by purchasing the FireDAC Add-On Pack). The deployment versions of EMS servers are ISAPI DLL, which run in conjunction with IIS (Microsoft's Internet Information Services). In this section you will learn how to configure IIS 7 to enable the EMS ISAPI DLLs, and how to configure your client applications to access this version of the server. As a bonus, you will learn how to access this ISAPI DLL from JavaScript and jQuery, permitting you to build web sites that get their data from your EMS endpoints.

As software developers, we probably spend as much time debugging and maintaining our software as designing it. In this next section you will learn how to step through your custom endpoints in your EMS packages, both when using the EMS development server, as well as with the ISAPI DLL version.

In the final section of this white paper you will learn how EMS compares to another Embarcadero middle-ware solution – DataSnap.

Before we continue, I want to emphasize that this paper is about EMS technology, how to implement it, and how to access those implementations. It is not, despite the title, about developing mobile solutions but rather how to use EMS to enable remote data

access and application logic. Even the name of this technology, Enterprise Mobility Services, could imply a mobile-only technology, which seems limited given the wide access EMS can provide to other clients.

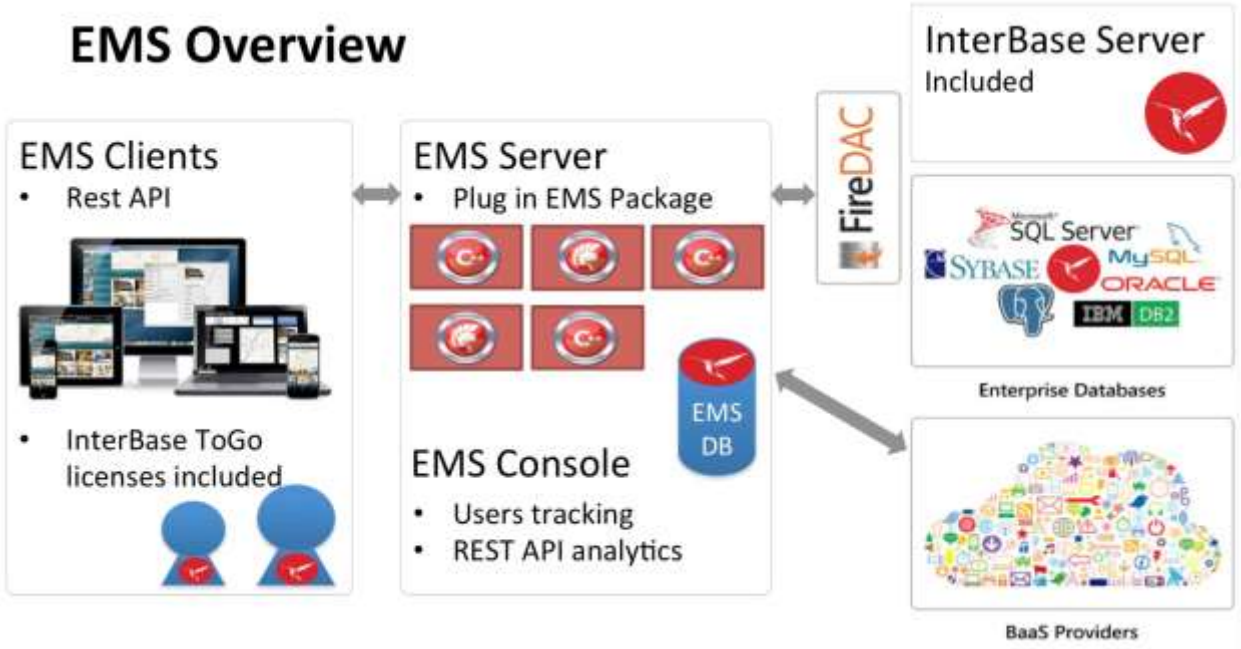
In truth, EMS is about highly accessible data, whatever the platform. It's about thin clients, those that do not, and in many cases, cannot, access enterprise data directly. In those cases, those thin clients must go through some sort of intermediary. EMS is such a solution, using a REST web service as the universally accessible platform for bi-directional data movement and remote procedure calls.

Because this paper is about accessing EMS data from client application in general, I have not focused specifically on mobile clients to demonstrate this access. Yes, the Delphi application that I use to demonstrate EMS data access can be compiled and deployed to an iOS or Android phone or tablet. However, this demonstration project was not designed as a mobile application, as these types of applications have user interface requirements and limitations that are beyond the scope of this paper. In other words, the techniques for accessing EMS endpoints that I demonstrate can and should be used in mobile applications, but I ask that you look to other white papers and video presentations on the Embarcadero site for details about mobile application user interface design.

# Overview of Enterprise Mobility Services (EMS)

Enterprise Mobility Services is an extensible, turnkey, middle-ware platform for securely exposing data and features to a wide range of client applications. These data and features are exposed through a REST interface that supports authentication and authorization and encrypted client/server communication. EMS client applications can be traditional desktop applications, mobile applications, web sites and web services, and any other type of application that can consume REST services.

The following figure, which was taken from RAD Studio's online help, depicts a general overview of the EMS technology stack. At the heart are the EMS server and the EMS console, depicted in the middle rectangle of this diagram.




The EMS server is a REST web service that exposes a collection of pre-defined REST endpoints that permit the management of users and groups. These users and groups are stored in InterBase database, and an InterBase license for this database is included with the EMS license.

The EMS console is a web service that exposes essential statistics about the EMS database and EMS server usage. For example, the EMS console lets console users view the server's users and groups. The EMS statistics exposed by the EMS console include the number of unique user visits, number of REST API calls and their distribution among the available REST endpoint. These data can be summarized by hour, day, month, or year, giving administrators insight into how the EMS server applications are being used.

The EMS server can be extended by adding one or more EMS packages, modules that can be created using RAD Studio's Delphi or C++Builder languages. It is through these custom endpoints that you expose your organization's data and operations. For example, you can create a set of custom endpoints that permit your mobile clients to request data from your company's database. Another set of endpoints can permit these clients to send updates back to the EMS server, where your custom code can evaluate these changes, and selectively post the changes back to the central database.

Since these custom endpoints are written in C++ or Object Pascal, they can use all of the power of RAD Studio to perform their necessary tasks. This is represented by the arrows between the EMS Server and the entities in the right-hand rectangle in the preceding diagram. For example, you can use FireDAC, or any of the other data access mechanisms in RAD Studio to communicate with your enterprise database servers. And

if you don't already have a SQL database server to store the data for your EMS applications, you can use the InterBase license that is included with EMS.

But it's not just databases that we are talking about. RAD Studio supports all of the standard protocols for network communication, and also includes a wide range of components that simplify your use of industry standard services. Your custom endpoints can provide your EMS clients with a gateway to all of your data, wherever it resides, whether it be in the cloud, inside your firewall, or even in local files on your servers.

Regardless of where your data is stored, your EMS server provides a secure gateway to you data. Your EMS server supports authentication and authorization, out of the box, along with SSL (Secure Socket Layer) encryption of your client/server communications.

But the real value of your EMS server comes from its clients. Because EMS exposes its endpoints using an industry standard REST API, any platform can host EMS clients. Using RAD Studio, you can build native mobile applications for iOS and Android, and these can use EMS client components to effortlessly interact with your EMS server. In addition, if you need secure local storage on your mobile devices, no problem. EMS includes licenses for InterBase ToGo, a small-footprint, transaction-supporting database server that can be installed on your mobile devices, and which supports encrypted storage.

You are not limited, however, to mobile applications. Your EMS client applications can be interactive desktop applications, Internet services, web services, console applications, in short, any type of application that can issue and consume HTTP requests.

Neither are you limited to EMS clients written in RAD Studio. For example, web pages employing JavaScript can access your EMS endpoints, so long as they abide by the security requirements of your EMS server, such as providing a valid username and password before being granted access to other EMS endpoints. In addition, logged in users will only be able to access the endpoints to which they are granted access rights. But that is a configuration issue, and you will learn how to do that later in this paper.

Taken together, EMS provides you with a secure platform from which you can access any data from anywhere.

# Setting Up EMS

In this section you are going to learn how to get started with EMS using the EMS development server that comes with RAD Studio. Later in this paper you will learn how

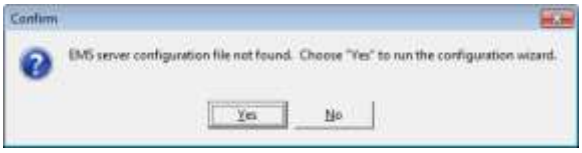
to install the production EMS server, which in this release is an ISAPI DLL that runs under IIS.

There are three pieces of the EMS development server that you will work with. The first is the EMS development server itself, and it is an HTTP server that you launch by running EMSDevServer.exe. The second is the EMS development console, which is also an HTTP server. You launch the EMS development console by running EMSDevConsole.exe. The third is emsserver.ini, the configuration file used by both the EMS development server as well as the EMS development console. This configuration file does not exist initially, but will be created the first time you run one of the EMS development servers.

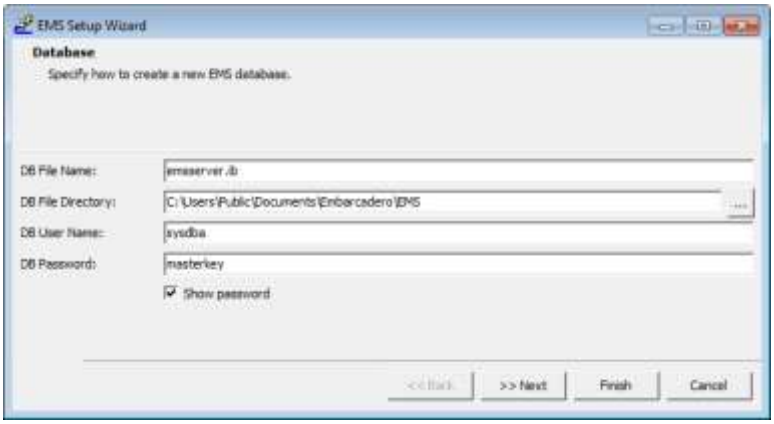
There are two versions of these EMS development servers, a 32-bit Windows version and a 64-bit Windows version. The 32-bit Windows version can be found in the bin folder, under the RAD Studio installation, and the 64-bit version is found in the bin64 folder, which is parallel to the bin directory. By default, RAD Studio XE7 is installed in the following folder:

```
C:\Program Files (x86)\Embarcadero\Studio\15.0
```

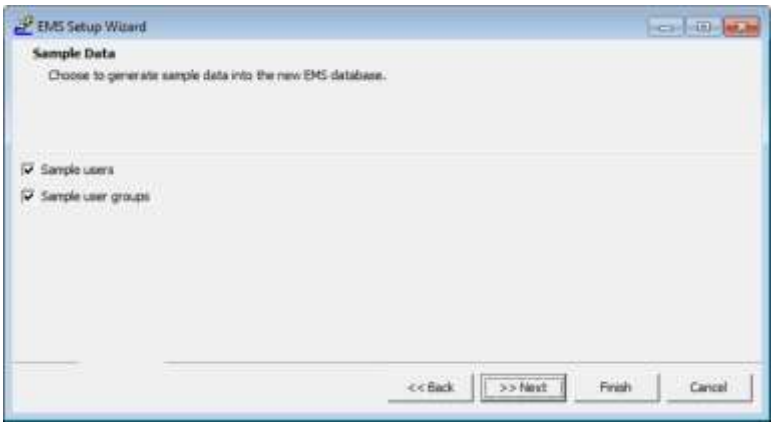
The first time you run either the EMS development server or the EMS development console, it will ask you if you want to run the EMS Setup Wizard. This wizard creates the EMS database that stores users and groups. It also creates the EMS configuration file:




Select Yes to continue to the Database page of the EMS Setup Wizard, shown in the following figure:

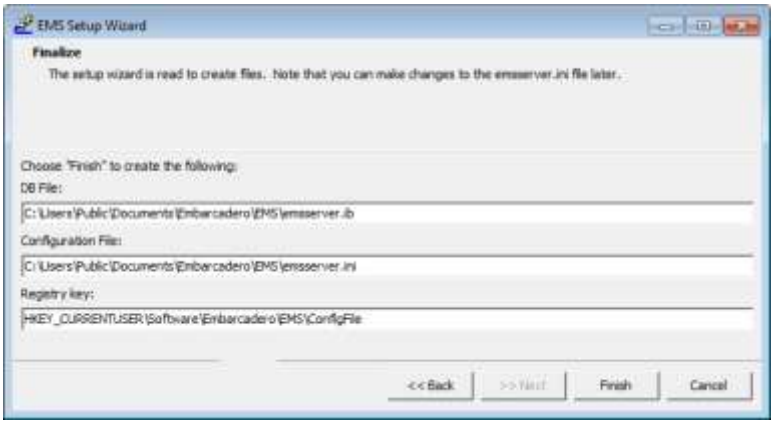


The first step in the EMS setup wizard is to create the EMS database. This database is an InterBase database that will be accessed by your EMS server. Because this is the development database, we can accept the default settings, which will place a database named emsserver.db, as well as the EMS configuration file, in the Embarcadero folder under your Documents library. When you are creating your production database, define the name and location of the database, and choose an administrative user name and password that is difficult to guess. Click Next to continue to the Sample Data page of the EMS Setup Wizard, shown in the following figure:




Use this page of the EMS Setup Wizard to choose whether or not to create a sample user and a group into which this user will be added. Again, since this is our development database, it would be nice to start with a user and a group for that user. When creating your production EMS database, you will want to uncheck these options, and create all of your users and groups explicitly. Click Next to continue to the Console page of the EMS Setup Wizard:

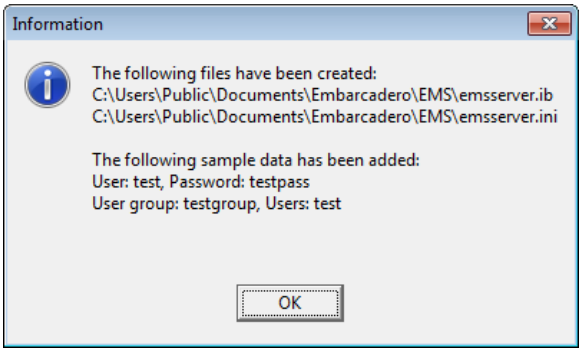

This page allows you to define the user name and password for the EMS console. These values can be changed later by editing the EMS configuration file. Click Next to proceed to the final page of the EMS Setup Wizard:




Use this screen to write to the Windows registry the location of the EMS database and configuration file. Unless you move the EMS configuration file to the same directory in which the EMS development server and EMS development console executables reside, they will use the provided registry key to locate the EMS configuration file. When using the deployment EMS server ISAPI DLL, your EMS configuration file will be located in the same directory as the ISAPI DLL, so this setting will be ignored. Go ahead and leave these values as they are, and click Finish to initiate the installation.

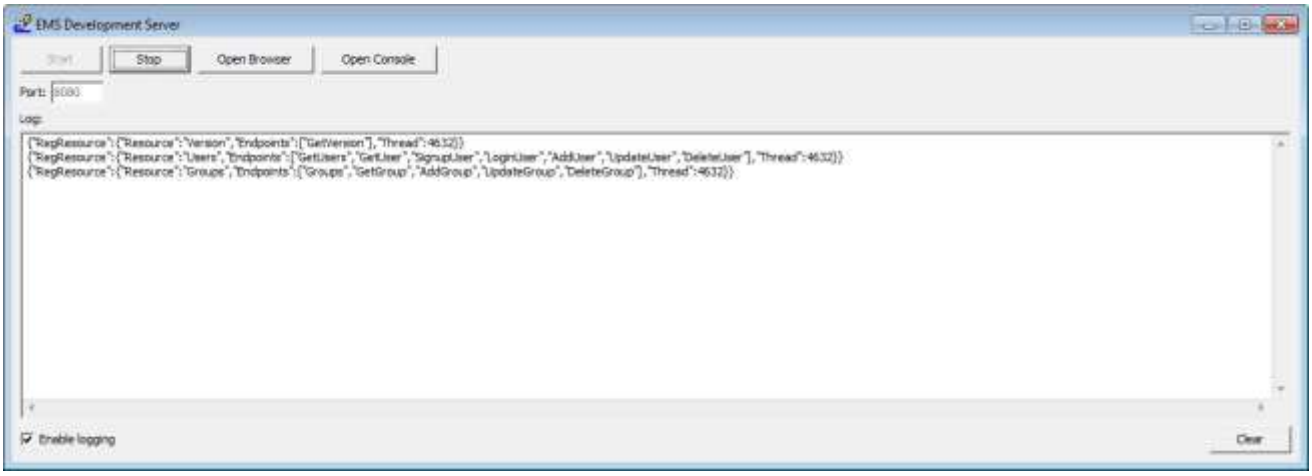
After a moment the EMS Setup Wizard will reports its success, displaying the following dialog box with information about your installation. Since we asked the EMS Setup Wizard to create a user and a group, is has created a user named test, which uses the password testpass. In addition, it has created a group named testgroup, and has added the user test to this group.

Before closing this dialog box, note the location of the emsserver.ini file. You are going to edit this file later in order to configure your EMS development server. Click OK when you are done:




Your EMS development server is now running. As mentioned before, it is a stand-alone HTTP server, and it is configured to listen on port 8080 (you can change this by editing the EMS configuration file). When this server started, it registered three built-in resources and their REST endpoints. By default, the Enable logging checkbox is

checked, and the results of these registrations appear in the displayed pane, as you can see in the following figure:




You can start and stop this server by clicking the Start and Stop buttons. Note, however, that if you make changes to the EMS configuration file, you need to terminate the server, and launch it again. Only when the server is launching does it read the EMS configuration file.

We could run the EMS development console at this point, but very little has happened. I will discuss the EMS console later in this paper. For now, let's start by learning how to interact with our EMS development server.

# Calling the EMS Administrative REST API

EMS is a turnkey middle-ware server, which is to say that it has functionality right out of the box. This functionality is provided by three built-in resources: Version, Users, and Groups. These resources expose REST endpoints that you can call using any client capable of making HTTP calls of the appropriate type for a particular endpoint.

The upcoming discussion of EMS client/server interaction assumes that you are already somewhat familiar with HTTP, REST, and to some extent, JSON (JavaScript Object Notation). If these technologies are new to you, the following section should give you enough information to get started. On the other hand, if you are already comfortable with REST and JSON, you can skip ahead to the section *Calling EMS Endpoints Using a Browser.*

# REST: A Brief Overview

HTTP, the HyperText Transfer Protocol, is the standard protocol for communication between an HTTP server (a web server) and an HTTP client, the most familiar of which is the common web browser. These three elements provide the foundation for the World Wide Web.

HTTP is a request-response protocol. HTTP clients issue HTTP requests to an HTTP server, which in turn takes an action based on the request. In the case of the web browser/web server interaction, the action is to either forward the request to another server, or to return a response, which typically contains HTML (HyperText Markup Language). The request includes a resource, which is defined by the URI (Universal Resource Identifier). The request also includes a header, which contains information about the request, and optionally a message body, in which relevant data can be passed.

The response includes a status, such as HTTP/1.1 200 OK, as well as headers that identify something about the response, such as the time at which the response was emitted, the type of web server that handled the request, cookies, and the like. The response may also include an optional body, which in the case of a web page, defines the head and body elements of the HTML response. This body, if present, always follows the last header after one blank line.

HTTP requests can take a number of different forms, and these are defined by the HTTP method type. When you enter a URL (Universal Resource Locator) in a browser, you are making an HTTP GET request. In an HTTP GET request, data is sent as part of the URL, and details about the browser are sent in the HTTP header.

Imagine that you want to view the Embarcadero Technologies home page on your browser. To do so, you might enter the following URL in your browser:

```
http://www.Embacadero.com
```

Assuming that you've issued this command from Internet Explorer version 10, the raw request looks something like the following:

```
GET http://www.embarcadero.com/ HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64;
Trident/6.0)
Accept-Encoding: gzip, deflate
Host: www.embarcadero.com
```

```
DNT: 1
Connection: Keep-Alive
```

GET identifies the request as an HTTP GET, and http://www.embarcadero.com is the URL, HTTP 1.1 is the protocol. The remainder of the text are the header entries, and they identify the version of the browser, as well as what this browser expects.

REST relies on the request-response nature of HTTP to make requests of software that lies on the web server. Specifically, a REST request references a resource on the web server to which the web server passes the request. This software reads the request, creates a response, and then returns this response to the web server, which in turn sends the response to the requesting client.

While many web server extensions, the software to which the server passes the request, return HTML for rendering in a browser, most REST web server extensions return data, most often in a standard format such as JSON or XML (eXtensible Markup Language). In addition, the HTTP clients for this data are usually programs that you write, and they will use that data for whatever purpose it was intended. EMS currently supports the creation of extensions for IIS, and these are in the form of ISAPI DLLs. EMS clients, on the other hand, can be any applications that can issue HTTP requests and consume the response.

Let's now focus exclusively on REST. A REST request includes the IP address or domain name of the server to which the request is sent (and sometimes a port number on which that server is listening), a resource on that server (the ISAPI DLL) to which IIS is going to delegate the response. The URL may also include additional data, which the resource can use to produce its response.

Recall that the purpose of REST, as far as we are concerned, is to permit a client application to interact with middle-ware, permitting that client to do what needs to be done. And what client applications typically need to do includes one or more of the following operations: reading data, writing data, updating data, and deleting data. These operations are often referred to using the acronym CRUD, which stands for Create, Read, Update, and Delete.

REST permits client applications to request these four types of operations, and they more or less map to HTTP method types. For example, an HTTP GET request corresponds to a read operation, while an HTTP DELETE has the obvious link to delete operations. POST operations correspond to create, and PUT operations are used to update data.

REST is not a rigid protocol, and there are no penalties, other than the mockery of your peers, for violating these rules. And in some cases, there are legitimate arguments for

straying from this basic approach. For example, many developers use PUT to either create or update a resource, one that is specifically defined. POST, by comparison, unconditionally creates a resource, even if that means that a second copy of the resource will be created.

How these various HTTP methods work is important to understand, once you begin to design your custom EMS endpoints. A GET request includes all information about the operation in the URL. The same thing applies to DELETE operations. If you are getting a particular element, such as a record for a particular customer, the customer identifier (presumably the value of the primary key for the associated table), is passed to the endpoint using a GET request. Your implementation of that endpoint reads the value, retrieves the record, and returns it in its response. A DELETE request is handled similarly, except that your implementation uses the key to remove, delete, or hide the corresponding resource.

PUT and POST operations are more involved. Specifically, these operations normally need to send more than an identifier to the REST server. For example, if you are using a PUT operation to create a new record, you are going to need to send the information that needs to be inserted. The same thing is true with respect to POST operations. As a consequence, operations that use PUT or POST pass the necessary data in the HTTP message body, which is capable of handling significantly larger amounts of data.

Finally, let's spend a few moments on JSON, a format that permits JavaScript objects (in an object-oriented sense) to be represented as text. A JSON object is a comma-separated list of name/value pairs enclosed in a matching pair of curly braces, where the name is the name of a property and the value is the value of that property. The following is an example of a JSON object that contains three properties:

```
{"One":"1","one":1,"TWO":true}
```

The names are case sensitive JSON strings, and must be distinct from one another within a given object, since a JavaScript object cannot have two different properties with the same name. The name is separated from its value by a colon. The value can be a number, string, true, false, null, array, or object. The JSON properties in the preceding JSON object where of type string, number, and true, respectively.

An array consists of a comma-separated list of JSON values, enclosed in a matching pair of square brackets. The elements of an array can all be the same JSON type, or they can be a mixture. As a result, a given array may include strings, objects, arrays, and so on.

The following is an example of a JSON object that contains a single property named Customers. This property is an array. The first element in the array is a JSON object, the second is an array of numbers, and the third is a null:

```
{"Customers":[{"Name":"Frank Borland"},[5.3,2,1000,-5.243e5],null]}
```

The JSON string type is a Unicode string enclosed in double quotation marks, where non-ASCII characters, as well as a handful of special characters, need to be escaped using a backslash, such as \r (carriage return), \" (double quote), \\ (backslash), \b (backspace), \t (horizontal tab), \u554A (啊), and so forth.

Numbers are not enclosed in quotes, and are used for both integers and floating point numbers. For floating point numbers, you use a period (.) as the decimal separator. You do not use thousands separators, but numbers can use exponential notation.

The remaining values, true, false, and null, are represented by the literals true, false, and null.

That's enough of a review. Let's start making calls to your EMS development server.

## Calling EMS Endpoints Using a Browser

You may recall that when the EMS development server started up, its logging displayed three resources that were loaded, along with their endpoints. A number of these endpoints are associated with GET requests, which means that it is possible to issue these request using a client as simple as a browser.

Before starting, you need to make sure that the EMS development server is running. Take a look at the EMS development server and make sure that the Start button is disabled. Also note the port on which this server is listening. The default value is 8080, and in all discussions of the EMS development server I assume that your server is using this port.

Now, open up a browser and enter the following URL:

```
http://localhost:8080/Version
```

Your browser should now look something like that shown in the following figure. I used Google's Chrome, so your screen might look a little different if you used a different browser:


Here is the HTTP request that my browser submitted to the EMS server:

```
GET http://localhost:8080/Version HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64;
Trident/6.0)
Accept-Encoding: gzip, deflate
Host: localhost:8080
DNT: 1
Connection: Keep-Alive
Cookie: EMSConsoleUser=consoleuser; dateBrowsed=2015-03-26T05:00:00.000Z
```

And here is how the EMS server responded:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: application/json; charset=ISO-8859-1
Content-Length: 44
Date: Thu, 26 Mar 2015 20:12:41 GMT

{"version":"1.0","server":"Embarcadero EMS"}
```

Notice that the EMS server responded with a content type of application/json. The message body, which appears on the second line after the last header, contains a simple JSON object with two string properties.

If you are using Internet Explorer, you probably had a very different experience. What you probably saw looked a lot like this:


It turns out that Internet Explorer doesn't know, by default, what to do with content that is returned with a content-type header of application/json, and that is the content type returned by the EMS server.

You can permit Internet Explorer to display JSON text by adding a key to your Windows registry that instructs Internet Explorer to open and display JSON content.

To do this, create a simple text file that contains the following script (exactly as shown, with the exception that you can omit any line beginning with a semicolon), and saving it with the extension .reg, such as json_in_ie.reg. After backing up your Windows registry (don't blame me if you don't take this step), double-click this script to add a new key to the Windows registry that will permit Internet Explorer to display JSON:

```
Windows Registry Editor Version 5.00
; Tell IE to open JSON documents in the browser.
; 25336920-03F9-11cf-8FD0-00AA00686F13 is the CLSID for the "Browse in
place" .
;

[HKEY_CLASSES_ROOT\MIME\Database\Content Type\application/json]
"CLSID"="{25336920-03F9-11cf-8FD0-00AA00686F13}"
"Encoding"=hex:08,00,00,00
```

*Thanks for this script goes to Cheeso on StackOverflow. This script was adapted from the one he posted in response to the question, "How can I convince IE to simply display application/json rather than offer to download it?".*

Let's try another EMS server endpoint. Enter the following URL in your browser:

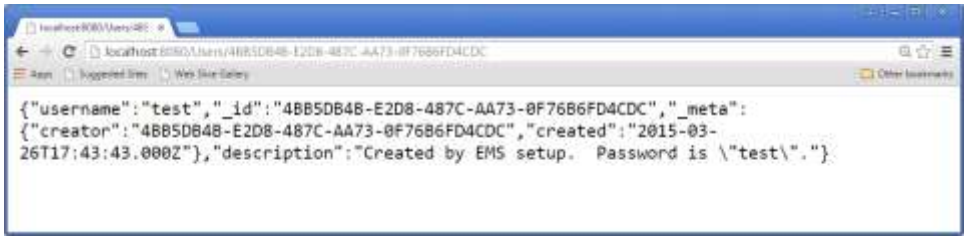
```
http://localhost:8080/Users
```

Your browser probably looks something like that shown in the following figure:


Both this and the preceding GET calls was handled by the default GET endpoint from the specified resource from the EMS development server, the resource being Version in the first call and Users in the second. These endpoints required no additional data. There are some GET requests, however, that do need additional data. For example, there is a GET call in the Users resource, GetItem, that requires additional information. If you want to get information about just one user, you need to pass that user's ID in the URL. To demonstrate this, select the value of the test user's ID in your browser and copy it to the clipboard. Next, add a forward slash after Users in the address line and then paste this id following the slash. In my case, the new URL looks like the following:

```
http://localhost:8080/Users/4BB5DB4B-E2D8-487C-AA73-0F76B6FD4CDC
```

The result, shown in the following figure, may appear to be identical to the request for all users, but it is not. If you look carefully, the result in the preceding figure is an array of JSON objects, which would have included one JSON object for each user, had there been more than one. JSON arrays appear within a pair of square braces. The result in the following figure contains a JSON object, not a JSON array:




## Calling EMS Endpoints Using the REST Debugger

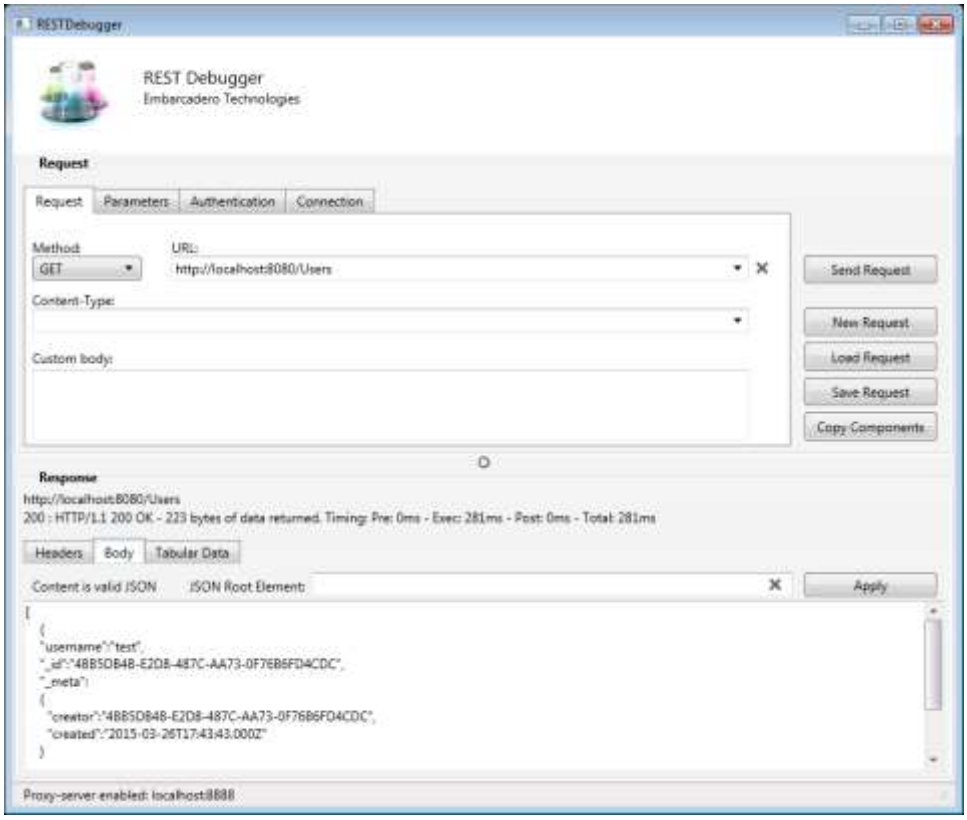
Let's face it — a browser can only take you so far, when it comes to testing your EMS server REST endpoints. Using the address line of a browser can only produce HTTP GET calls.

Fortunately, RAD Studio ships with a handy tool for testing REST interfaces, and it can handle all four of the HTTP method types we need. This tool is the REST Debugger, and you access it by selecting Tools | REST Debugger from RAD Studio's main menu.

Let's start with a simple GET statement. Enter the following URL into the URL fields of the Request pane:

```
http://localhost:8080/Users
```

Click the Send Request button. After the REST Debugger receives the response from the EMS development server, it displays the headers in the Response pane. If you click the Body tab in the response pane, you can see the array of JSON objects returned by this method, as shown in the following figure.




Now let's try something more complicated. Specifically, let's create a new user.

A new user is created by an HTTP POST. Begin by setting the Method dropdown of the Request pane to POST. Next, ensure that the URL field still contains the reference to the Users resource, as shown here:

```
http://localhost:8080/Users
```

As I mentioned earlier in this paper, a POST, like a PUT, passes data to the HTTP server in the message body. In this case we are POSTing a new user to the EMS database, and to do so we need to pass a JSON object consisting of two properties, where one property is named username and the other is password. So, in order to create a new user named *newuser,* and a password of *Delphi,* we would use a JSON object that looks like the following:

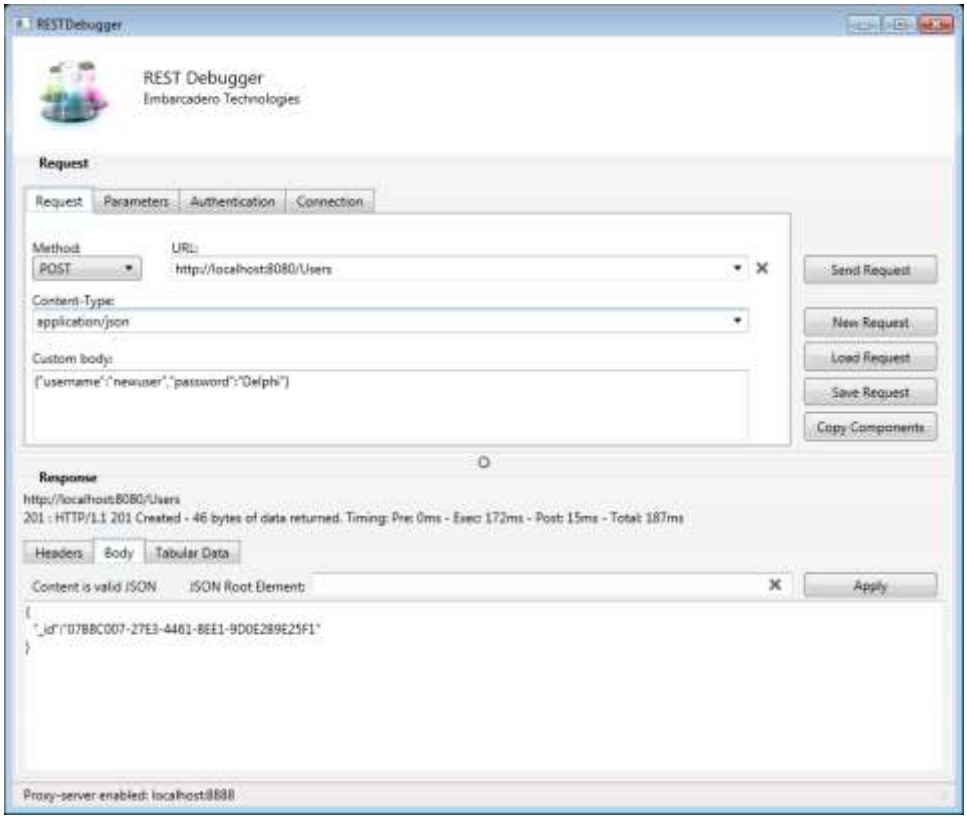
```
{"username":"newuser","password":"Delphi"}
```

Enter the above text into the Custom body memo on the Request pane of the REST Debugger.

Because we are sending JSON data, we also have to inform the EMS server about the type of data it is going to receive. To do this, enter the following data into the Content-Type field of the Request pane of the REST Debugger:

```
application/json
```

We are now set. Click Send Request to submit this POST request to the EMS developer server:




Notice in the preceding figure that the body of the HTTP response also contains a JSON object. This object is informing us of the value assigned to this new user's ID field. In addition, if you click on the Headers tab of the Response pane, you will see the headers of the corresponding HTTP response, which will look something like that shown here:

```
Connection=close
Content-Type=application/json; charset=ISO-8859-1
Content-Length=46
Date=Thu, 26 Mar 2015 21:12:45 GMT
Location=http://localhost:8080/Users/07B8C007-27E3-4461-8EE1-9D0E2B9E25F1
```

To see the newly added user, change the Method type in the Request pane back to GET and click Send Request. From the Body tab of the Response pane you will now see the new user, as shown in the following Body tab content:


```
[
    {
    "username":"test",
    "_id":"4BB5DB4B-E2D8-487C-AA73-0F76B6FD4CDC",
    "_meta":
    {
      "creator":"4BB5DB4B-E2D8-487C-AA73-0F76B6FD4CDC",
      "created":"2015-03-26T17:43:43.000Z"
    }
,
    "description":"Created by EMS setup.  Password is \"test\"."
  }
,

  {
    "username":"newuser",
    "_id":"07B8C007-27E3-4461-8EE1-9D0E2B9E25F1",
    "_meta":
    {
      "creator":"07B8C007-27E3-4461-8EE1-9D0E2B9E25F1",
      "created":"2015-03-26T21:27:47.000Z"
    }
  }
]
```


# Invoking the EMS Administrative REST API Using RAD Studio Components

If you build your EMS clients using RAD Studio, you will find a handful of components that simplify your communication with the server's administrative REST API. For example, if you build your mobile apps using the FMX (FireMonkey) framework and RAD Studio's native compilers for Android and iOS, communicating with your EMS server is a snap.

In this section I am going to introduce you to four RAD Studio components: EMSProvider, BackendUsers, and BackendGroups, and BackendQuery. There is another essential component that you will need for most of your applications, called BackendAuth, but we'll cover that one after we discuss authentication and authorization.

The EMSProvider is a connection component whose properties identify the EMS server to which you are connecting. Your backend components, in turn, will reference this

shared EMSProvider in order to identify the EMS server to which they will send REST requests.

EMSProvider has four essential properties: URLBasePath, URLHost, URLPort, and URLProtocol. URLHost and URLPort are pretty simple, in that they point to the server and the server's port on which the HTTP server is listening. URLBasePath is used to reference the EMS ISAPI DLL, including its virtual directory path. When you are using the EMS development server, you leave this property blank. Use URLProtocol to select which protocol you will use to connect to the server, which will be either http or https.

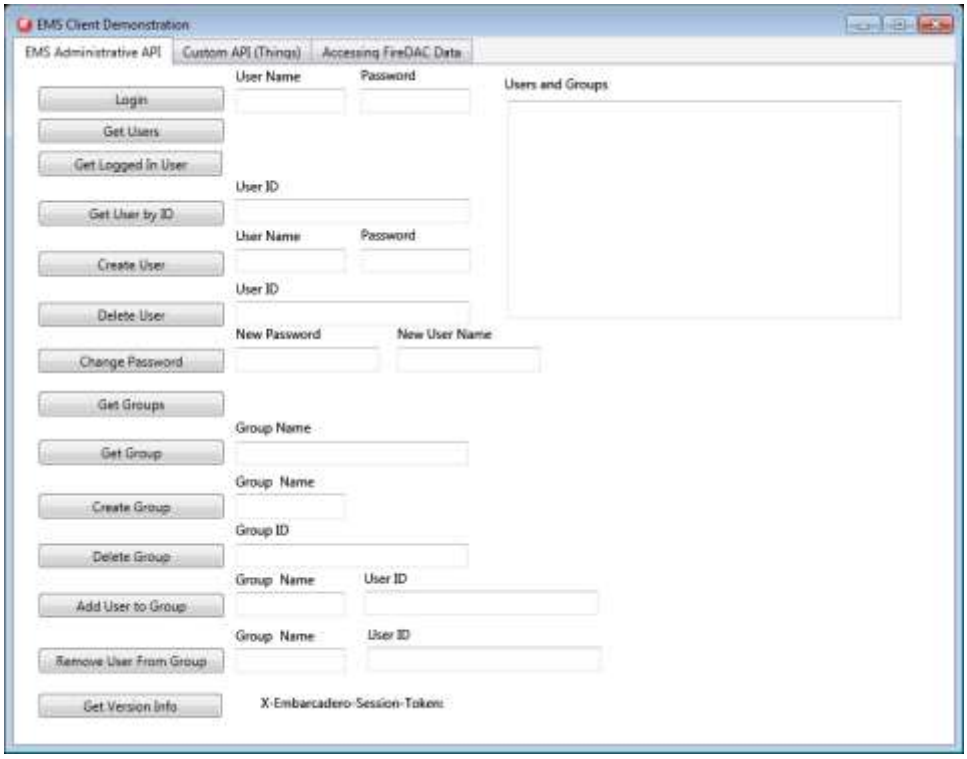
If you are using a proxy server, you use the ProxyPort and ProxyServer property of the EMSProvider to identify this server. If that proxy server requires a user name and password, there are properties for those as well.

The BackendUsers and BackendGroups components are pretty self-explanatory. You use the BackendUsers component to make requests to the Users resource of the EMS server, and the BackendGroups component to make requests to the Groups resource. There are only two properties that you normally set on these components. You assign an EMSProvider to their Provider property. And, if you are using authentication and authorization, you will assign a configured BackendAuth component to the Auth property.

BackendUsers and BackendGroups have one limitation, as far as I can tell. They cannot perform the simple GET call on their resource. For example, the BackendUsers component cannot get a list of users. That is where the BackendQuery comes in.

In addition to BackendUsers and BackendGroups, you can also use BackendQuery, which can also be used to retrieve information from the Users and Groups resources. You configure BackendQuery similar to how you configure the BackendUsers and BackendGroups components. You set the Provider property, and, if necessary, the Auth property. In addition, you set its BackendService property to the name of the resource it is querying. There are two BackendQuery components in the demonstration project, and both are wired to the same EMSProvider, and they are used to get the list of users and groups, respectively. The BackendQueryUsers component has BackendService set to Users, and BackendQueryGroups has BackendService set to Groups.

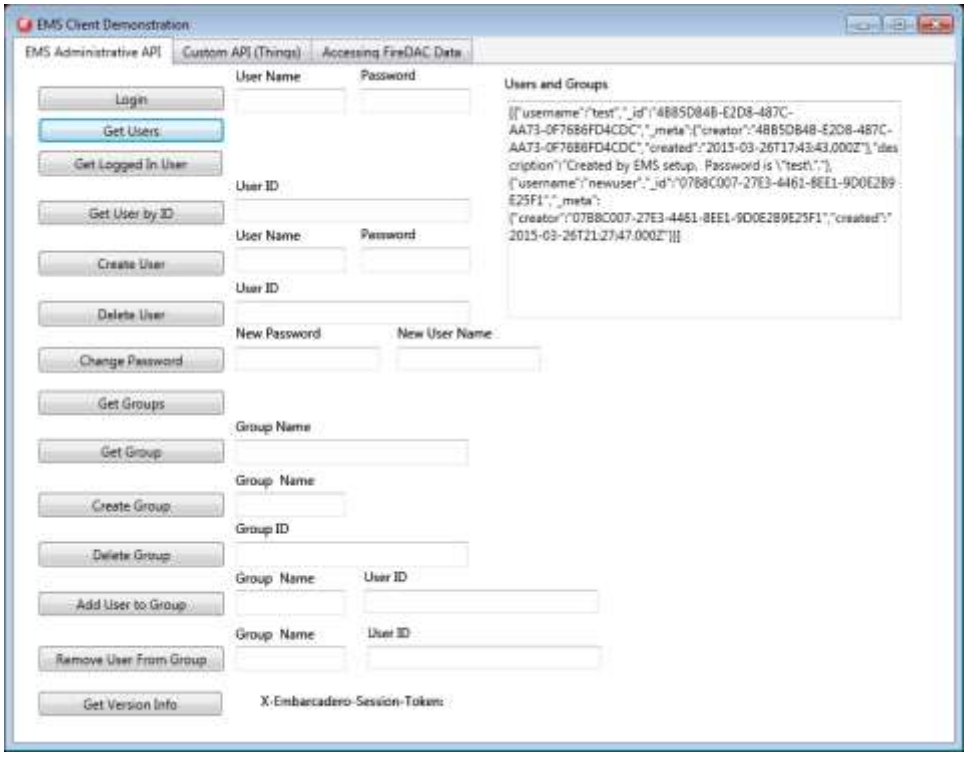
The use of these components, as well as many others, is demonstrated in the EMSDemoClient project, whose main form is shown in the following figure. This project is an FMX project, and can be compiled and run in any of the operating systems supported by RAD Studio.




Let's get some of the simplest calls out of the way first. As mentioned above, you use a BackendQuery to get your Users and your Groups. You do this by calling the Execute method, after which you retrieve the resulting JSON object from the JSONResult property. This is demonstrated in the following code, which is associated with the button labeled Get Users. Similar code is used to list the available groups. Note that I have placed all of the REST components, such as BackendQuery and EMSProvider, on a separate data module. The dm in the following code is a reference to the data module on which these components reside:

```
procedure TForm1.GetUsersBtnClick(Sender: TObject);
begin
  Memo1.Lines.Clear;
  dm.BackendUserQuery.Execute;
  Memo1.Lines.Text := dm.BackendUserQuery.JSONResult.ToJSON;
end;
```

The following figure shows how the main form looks after the Get Users button is clicked:




Getting information about a particular user, creating a user, and deleting a user (and the corresponding calls to the Groups resource), are somewhat more complicated in that they require that you pass information to the EMS server. As you recall, that information must be passed in the form of a JSON object. Fortunately, the BackendUsers and BackendGroups components simplify this process. Specifically, instead of you having to create a JSON object, you simply pass the necessary data to the appropriate method, and the implementation of that method takes the responsibility for the creation of the associated JSON object.

Here is the code assigned with the buttons labeled Get Group, Create Group, and Delete Group. There are similar implementations for the corresponding user buttons:

```
procedure TForm1.GetGroupBtnClick(Sender: TObject);
var
  group: TBackendEntityValue;
  val: TFindObjectProc;
begin
  dm.BackendGroups1.Groups.FindGroup(GroupNameEdit.Text, GroupFound);
end;

procedure TForm1.CreateGroupBtnClick(Sender: TObject);
var
  group: TBackendEntityValue;
begin
  dm.BackendGroups1.CreateGroupsAPI.CreateGroup(CreateGroupNameEdit.Text,
                                                TJSONObject.Create,
group);
```

```
end;

procedure TForm1.DeleteGroupBtnClick(Sender: TObject);
begin
  dm.BackendGroups1.Groups.DeleteGroup(DeleteGroupEdit.Text);
end;
```

Neither the Create Group nor the Delete Group event handlers really need to do anything with the response that is returned by the server. The same cannot be said about the Get Group method, which will return a JSON object containing the requested group's information. If you inspect the GetGroupBtnClick method, you will notice that FindGroup is passed two parameters. The first parameter is the name of the group. The second parameter, however, is a reference to a method (and which could have contained an anonymous method if you were so inclined). This method, named GroupFound, is executed when the call to GetGroup is complete, and is passed the returned JSON object in one of its parameters. The implementation of GroupFound is shown here:

```
procedure TForm1.GroupFound(const AObject: TBackendEntityValue;
  const AJSON: TJSONObject);
begin
  Memo1.Lines.Clear;
  Memo1.Lines.Add('Group name: ' + AObject.GroupName);
  Memo1.Lines.Add('Created: ' +
    FormatDateTime('mmm dd, yyyy hh:mm', AObject.CreatedAt));
  Memo1.Lines.Add('Updated: ' +
    FormatDateTime('mmm dd, yyyy hh:mm', AObject.UpdatedAt));
end;
```

There is more to the Users and Groups resources than I've covered here, but you should have a good idea of how these backend components are used. We'll return to the administrative REST API later in this paper when we address the issue of EMS security.

# Extending EMS with Custom Resources and Endpoints

The EMS server supports users, groups, authentication, and authorization right out of the box, which relieves you from having to build your own mechanisms for these important functions. However, this is basic infrastructure that is necessary to support something else. That something else is the data and operations of your organization.

You extend your EMS server using runtime packages that define and register resources that you define. Each resource can include one or more endpoints, and these endpoints can be invoked from your REST client applications.

You can build your runtime packages manually, as well as write the classes that define your resources and expose your endpoints, but there is no need to do so. RAD Studio provides you with two wizards in the Object Repository that take care of most of the groundwork, greatly simplifying the creation of your custom REST API. One of these wizards creates the basic runtime package, which can optionally include one resource. The second wizard adds a new resource to an existing EMS runtime package.

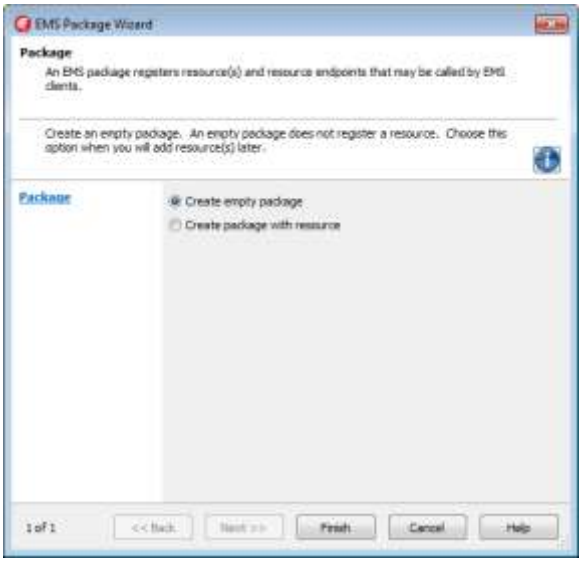
## Creating an EMS Package

The following steps demonstrate how easy it is to create a custom REST API that you can access through your EMS server:

1. Create the new runtime package by selecting File | New | Other to display the Object Repository.
2. From the EMS tab of the Object Repository, double-click the EMS Package icon to launch the EMS Package Wizard.

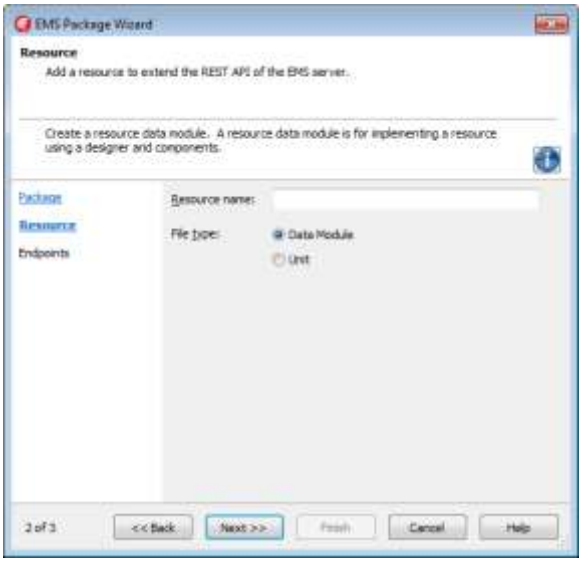
EMS Package

3. The Package page of the EMS Package Wizard is shown in the following figure:



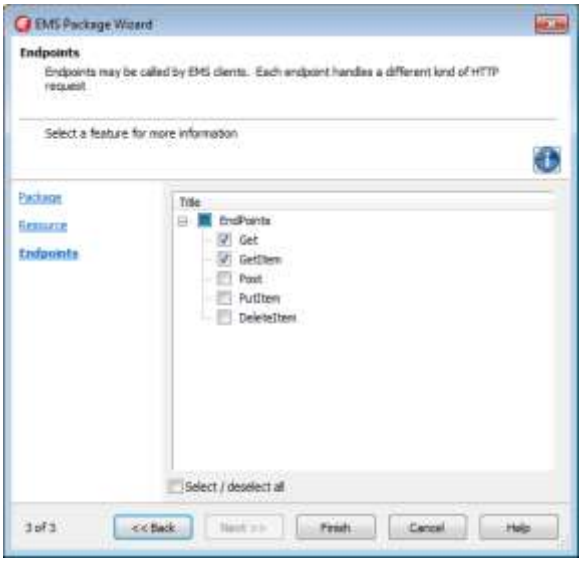

4. On the Package page of the EMS Package Wizard you can choose to create an empty package, to which you can later add one or more resources, or you can create a package with one new resource. If you select to create an empty package, you are done. But for our purposes, select the Create package with

resource radio button, which will enable the Next button. Click Next to move to the Resource page of the EMS Package Wizard, shown in the following figure:




5. You use the Resource page of the EMS package Wizard to define the name of your resource, as well as whether to implement it as a data module or a TObject descendant. The resource name will be part of the URL that REST clients will use to access its endpoints, so you want a name that is descriptive. Note, however, that you can easily change the resource name at a later time, so you shouldn't fret about it too much. For this demonstration, enter the resource name Demo.

   The choice of file type will depend on what you want to expose through your endpoints. If you are going to access a database from your endpoints you will want to use a data module, since data modules support a design surface, which will permit you to place and configure non-visual components, such as FireDAC data access components, at design time. If you do not need to configure any design-time components, it doesn't matter. Personally, I prefer to have a data module since it is more flexible. Select the Data Module radio button and click Next to advance to the Endpoints page of the EMS Package Wizard, shown in the following figure:




6. The endpoints that appear on the Endpoints page of the EMS Package wizard correspond, for the most part, to the GET, PUT, POST, and DELETE HTTP methods discussed earlier in this paper. As a result, you should choose to generate the endpoints that represent the capabilities you are exposing to the REST clients.

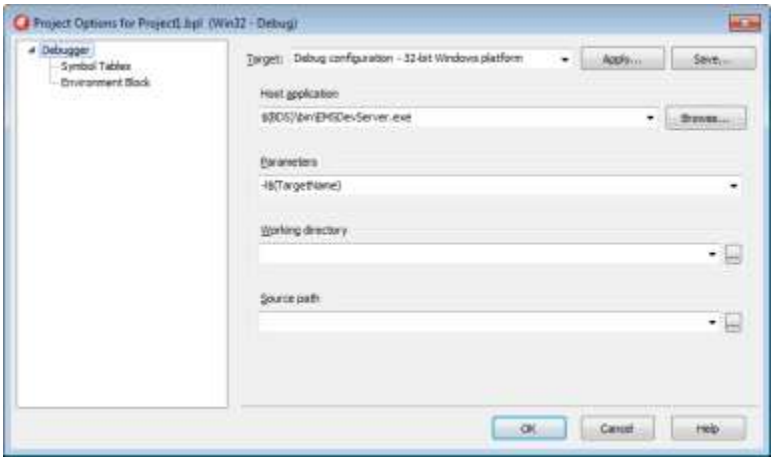
   In keeping with the spirit of REST APIs, Get is designed to return the entire resource, and GetItem will return an element of the resource, based on an identifier submitted by the client in the request. Post submits a resource, PutItem either inserts or updates an element, and DeleteItem deletes an element of a resource. In reality, you can do anything you want with these various endpoints, so long as you pass data between the server and clients in a manner consistent with the HTTP method type (for example, passing data to PutItem in the HTTP message body). Nonetheless, consistency with the spirit of REST will not make you any enemies.

   As is the case with the Resource name, you can easily add or remove these endpoints at a later time, and you can add additional methods, beyond what is offered here. Since we're using this package for demonstration purposes, let's go ahead and place a check mark next to all of the endpoint options. Complete the EMS Package wizard by clicking Finish.

There are a number of interesting features of the package that is created by the EMS package wizard. One is that it is created as a runtime package by default. By comparison, packages created by the Object Repository's Package Wizard are designtime and runtime packages by default.

The second feature of this package is that its project is configured, by default, to launch the EMS development server if you run it. A package is a DLL, and must be loaded by

another process. Normally, you cannot run a package or DLL from the RAD Studio IDE. However, if a host application has been defined, clicking Run or Run without debugging launches the host application. You can see that the EMS development server has been defined as the host application for this project if you select Run | Parameters from the RAD Studio main menu:




# Understanding the EMS Resource and Endpoints

An EMS resource is a class that is decorated with the ResourceName attribute, to which is passed the name of the resource. This is the name that appears in a REST request URL, immediately following the URI portion that references the EMS server. In addition to the ResourceName attribute, this class must also be registered by a call to the RegisterResource method, which is declared in the EMS.ResourceTypes unit.

The following is the generated Register procedure as it appears in the newly generated package resource:

```
procedure Register;
begin
  RegisterResource(TypeInfo(TDemoResource1));
end;
```

We requested that our resource be generated as a data module, in case we wanted to use design-time, non-visual components in our endpoint implementations. The following is the class declaration that was generated by the EMS Package Wizard. This class includes five published methods, based on the selections we made from the Endpoints page of the EMS Package Wizard.

```
type
  [ResourceName('Demo')]
  TDemoResource1 = class(TDataModule)
```

```
  published
    procedure Get(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest;const AResponse:
TEndpointResponse);
    [ResourceSuffix('{item}')]
    procedure GetItem(const AContext: TEndpointContext;
      const ARequest:TEndpointRequest; const AResponse:
TEndpointResponse);
    procedure Post(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('{item}')]
    procedure PutItem(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('{item}')]
    procedure DeleteItem(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
  end;
```

You will notice that most, but not all, of the methods that appear in the resource class definition are decorated with the ResourceSuffix attribute. This attribute permits you to define specific endpoints within the resource, as well as the name of parameters that a particular endpoint requires. None of the ResourceSuffix attributes appearing in the generated resource include a more specific endpoint, but all of them identify a single parameter, named Item in this case. I'll discuss ResourceSuffix attributes that specify a more specific endpoint later in this paper in the section, *Creating Additional Custom Endpoints*.

Why do the Get and Post method not include a ResourceSuffix attribute? The simple answer is that Get, as implemented by the EMS Package Wizard, is not assumed to be passed any parameters, returning not a particular element but the entire resource. The same is true of Post, which in REST is often used to replace an entire resource, based on data passed to the EMS server in the HTTP message body.

Turning our attention to parameters in ResourceSuffix attributes, Item is identified as a parameter name due to its enclosure in curly braces. This name can be used in your endpoint implementations to retrieve the corresponding parameter from the REST request. This can be seen in the following code, which shows the implementations of the requested endpoints, as generated by the EMS Package Wizard:

```
procedure TDemoResource1.Get(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
end;

procedure TDemoResource1.GetItem(const AContext: TEndpointContext;
```

```
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  LItem: string;
begin
  LItem := ARequest.Params.Values['item'];
end;

procedure TDemoResource1.Post(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
end;

procedure TDemoResource1.PutItem(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  LItem: string;
begin
  LItem := ARequest.Params.Values['item'];
end;

procedure TDemoResource1.DeleteItem(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  LItem: string;
begin
  LItem := ARequest.Params.Values['item'];
end;
```

There is one more attribute that you can use in your resource definition, and it can be used to decorate individual published methods, similar to the ResourceSuffix attribute. This attribute is EndpointName, and it permits you to define an alias for the endpoint it decorates. This attribute is used by the EMS console to display the alias, as opposed to the actual endpoint name, when it reports on endpoint statistics. This alias is also used to configure endpoint access in the Server.Authorization section of the EMS configuration file.

## Defining Your REST API

As the old saying that goes, "The devil is in the details," and it applies so very well to the implementation of your REST endpoints. In short, so long as your REST clients conform to the signature of your REST endpoints, they can send almost anything to your EMS server, and your custom implementations can return almost anything. Of course, the "almost anything" that I am referring to must make sense to both the client and your endpoint implementation.

For example, if your resource is called Customers, you might specify that GetItem means that the client wants a single record from the customer table in your database,

and will identify the desired customer in the GetItem invocation by sending a value that identifies a customer, such as the customer's account number, in the GET request. In return, your GetItem implementation executes a query that retrieves a single customer based on the identifier it receives in the request, and packages that data up as a JSON object, which it then returns to the client.

Here is a sample of a simple, but recognizable REST API. Here is the Things resource definition, found in the ThingsModuleu unit of the EMSDemoPkg project:

```
type
  [ResourceName('Things')]
  TThingResource = class(TDataModule)
    FDConnection1: TFDConnection;
    FDQuery1: TFDQuery;
    FDQuery2: TFDQuery;
    FDGUIxWaitCursor1: TFDGUIxWaitCursor;
  public
    constructor Create(AOwner: TComponent); override;
  published
    procedure Get(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('{item}')]
    procedure GetItem(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    procedure Post(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('{item}')]
    procedure PutItem(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('{item}')]
    procedure DeleteItem(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
  end;
```

This resource provides REST clients with access to Things, which within the context of this example is a record that includes two values, ThingID and ThingName. The table that holds Things (MyThings), and a stored procedure that can insert or create a Thing, is defined in the overridden constructor of this resource. That constructor is only necessary to ensure that your database includes the Things table and the CreateOrUpdateThing stored procedure. In practice, your database will already be complete before you create your resources, so an override would not be necessary.

The Get function returns a list of all Things, and GetItem returns a specific Thing, based on a ThingID submitted with that request. PutItem is used to insert or update a Thing,

based on a ThingID and ThingName, and Post inserts one or more ThingNames, assigning a unique ThingID to each. (This difference between PUT and POST appears to be in the spirit of REST.) Finally, DeleteItem deletes a Thing based on ThingID.

Here are the implementations of this resource's methods along with a method used to generate a unique ThingID based on a GUID (globally unique identifier):

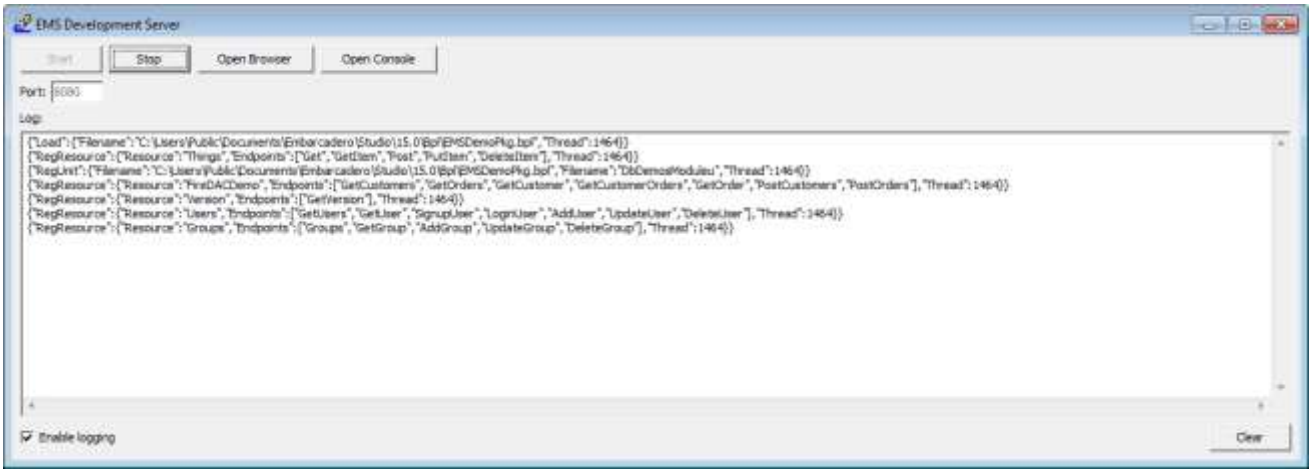
```
function GetGuid: string;
var
  GUID: TGUID;
begin
  CreateGUID(GUID);
  Result := GuidToString(GUID);
end;

procedure TThingResource.Get(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  ja: TJSONArray;
  jo: TJSONObject;
  FDQuery: TFDQuery;
begin
  FDQuery := TFDQuery.Create(nil);
  FDQuery.Connection := FDConnection1;
  FDQuery.SQL.Text := 'SELECT ThingID, ThingName FROM MYTHINGS';
  FDQuery.Open();
  try
    ja := TJSONArray.Create;
    while not FDQuery.Eof do
    begin

ja.AddElement(TJSONArray.Create(TJSONString.Create(FDQuery.Fields[0].AsSt
ring),
                    TJSONString.Create(FDQuery.Fields[1].AsString)));
      FDQuery.Next;
    end;
    jo := TJSONObject.Create(TJSONPair.Create('response', ja));
    AResponse.Body.SetValue(jo, True);
  finally
    FDQuery.Close;
  end;
end;

procedure TThingResource.GetItem(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  ja: TJSONArray;
  jo: TJSONObject;
  FDQuery: TFDQuery;
  ThingID: string;
begin
  ThingID := ARequest.Params.Values['item'];
  FDQuery := TFDQuery.Create(nil);
  FDQuery.Connection := FDConnection1;
```

```
  FDQuery.SQL.Text := 'SELECT ThingID, ThingName FROM MYTHINGS WHERE
ThingID = :id';
  FDQuery.Params[0].AsString := ThingID;
  FDQuery.Open();
  if FDQuery.RecordCount = 0 then
    AResponse.RaiseNotFound('Item not found', ThingID + ' not found')
  else
  begin
    ja :=
TJSONArray.Create(TJSONString.Create(FDQuery.Fields[0].AsString),

TJSONString.Create(FDQuery.Fields[1].AsString));
    jo := TJSONObject.Create(TJSONPair.Create('response', ja));
    AResponse.Body.SetValue(jo, True);
  end;
  FDQuery.Close;
end;

procedure TThingResource.Post(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  FDQuery: TFDQuery;
  jo: TJSONObject;
  ja: TJSONArray;
  jv: TJSONValue;
begin
  if ARequest.Body.TryGetObject(jo) then
  begin
    ja := jo.GetValue('request') as TJSONArray;
    FDQuery := TFDQuery.Create(nil);
    try
      FDQuery.Connection := FDConnection1;
      FDConnection1.StartTransaction;
      FDQuery.SQL.Text := 'EXECUTE PROCEDURE CreateOrUpdateThing( :id,
:name)';
      try
        try
          for jv in ja do
          begin
            FDQuery.Params[0].AsString := GetGuid;
            FDQuery.Params[1].AsString := TJSONString(jv).Value;
            FDQuery.ExecSQL;
          end;
          FDConnection1.Commit;
          AResponse.Body.SetValue(TJSONObject.Create(
                                  TJSONPair.Create('response',
                                    IntToStr(ja.Count) +
                                    ' thing(s) created')), True);
        except
          FDConnection1.Rollback;
          AResponse.RaiseBadRequest('Error', 'Invalid data. POST
aborted');
        end;
      finally
        FDQuery.Free;
      end;
    finally
```

```
      jo.Free;
    end;
  end
  else
    AResponse.RaiseBadRequest('Error', 'JSON expected in message body');
end;

procedure TThingResource.PutItem(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
vabr
  FDQuery: TFDQuery;
  jo: TJSONObject;
  ja: TJSONArray;
  jv: TJSONValue;
begin
  jo := TJSONObject.ParseJSONValue(ARequest.Params.Values['Item']) as
TJSONObject;
  try
    ja := jo.GetValue('request') as TJSONArray;
    FDQuery := TFDQuery.Create(nil);
    try
      FDQuery.Connection := FDConnection1;
      FDQuery.SQL.Text := 'EXECUTE PROCEDURE CreateOrUpdateThing( :id,
:name)';
      FDQuery.Params[0].AsString := ja.Items[0].Value;
      FDQuery.Params[1].AsString := ja.Items[1].Value;
      try
        FDQuery.ExecSQL;
      except
        AResponse.RaiseBadRequest('Error', 'Invalid data. PUT aborted');
      end;
     finally
      FDQuery.Free;
    end;
  except
    AResponse.RaiseBadRequest('Bad request', 'Bad request. Expecting
JSONArray');
  end;
end;

procedure TThingResource.DeleteItem(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  ThingID: string;
  FDQuery: TFDQuery;
begin
  ThingID := ARequest.Params.Values['item'];
  FDQuery := TFDQuery.Create(nil);
  try
    FDQuery.Connection := FDConnection1;
    FDQuery.SQL.Text := 'DELETE FROM MyThings WHERE ThingID = :id';
    FDQuery.Params[0].AsString := ThingID;
    FDQuery.ExecSQL;
  finally
    FDQuery.Free;
  end;
end;
```

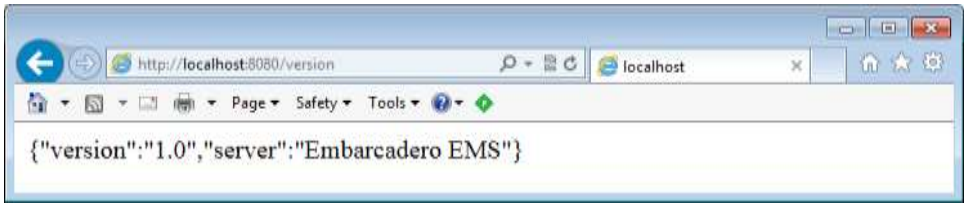
# Testing Your Endpoints

Load the EMSDemoPkg into the RAD Studio and run it without debugging. You can do this by either by clicking the Run without debugging button on the RAD Studio toolbar, or select Run | Run Without Debugging from RAD Studio's main menu. The running EMS development server is shown in the following figure:




Before going further, note that the EMS development server is logging its operations. The first entry in the log identifies that the custom EMS package was loaded, and that the Things resource was registered. Next it shows all registered resources and their endpoints. These resources include Things, and another resource that I will discuss a little later in this project, FireDACDemo. Finally, the resources of the administrative API are registered (Version, Users, and Groups).

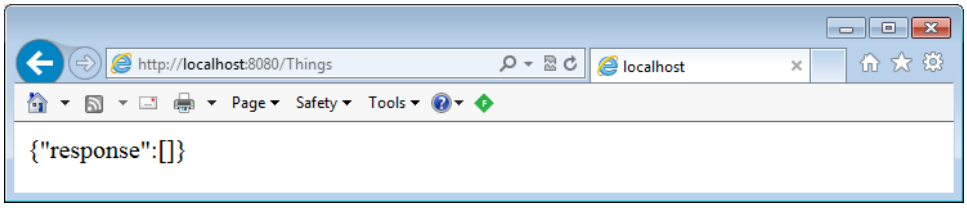
Click the Open Browser button to launch your default web browser, which issue a Get request for the Things resource, as shown here:




Now, modify the URL to look like the following:

```
http://localhost:8080/Things
```

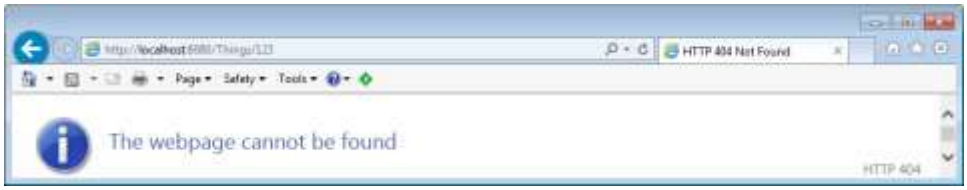
Once you instruct the browser to request this endpoint, the response will look something like the following:




This doesn't look like a real response, but it is. Specifically, Get returned an array of things, but there are no Things, since the MyThings table has only just been created. We have another GET URL we can try, which will invoke the GetItem endpoint. Let's try the following URL:

```
http://localhost:8080/Things123
```

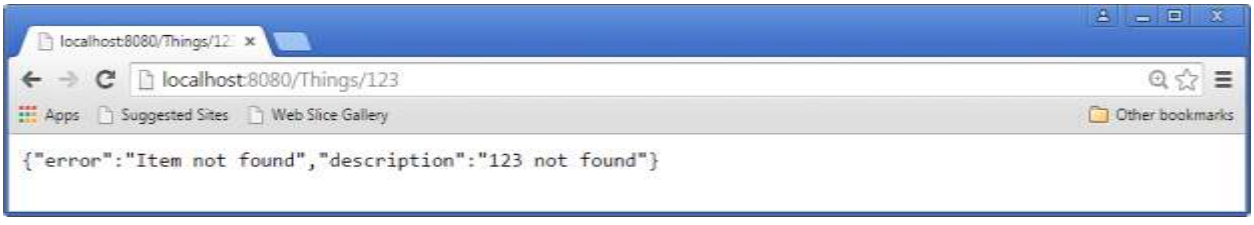
When I do this from Internet Explorer, I get the following response:




This, too, might not look like much, but it really is a reply from the Things resource. If you examine the implementation of GetItem, you will notice that when the query for a specific thing returns an empty result, the endpoint raises an exception. The line that raises this exception looks like the following:

```
AResponse.RaiseNotFound('Item not found', ThingID + ' not found')
```

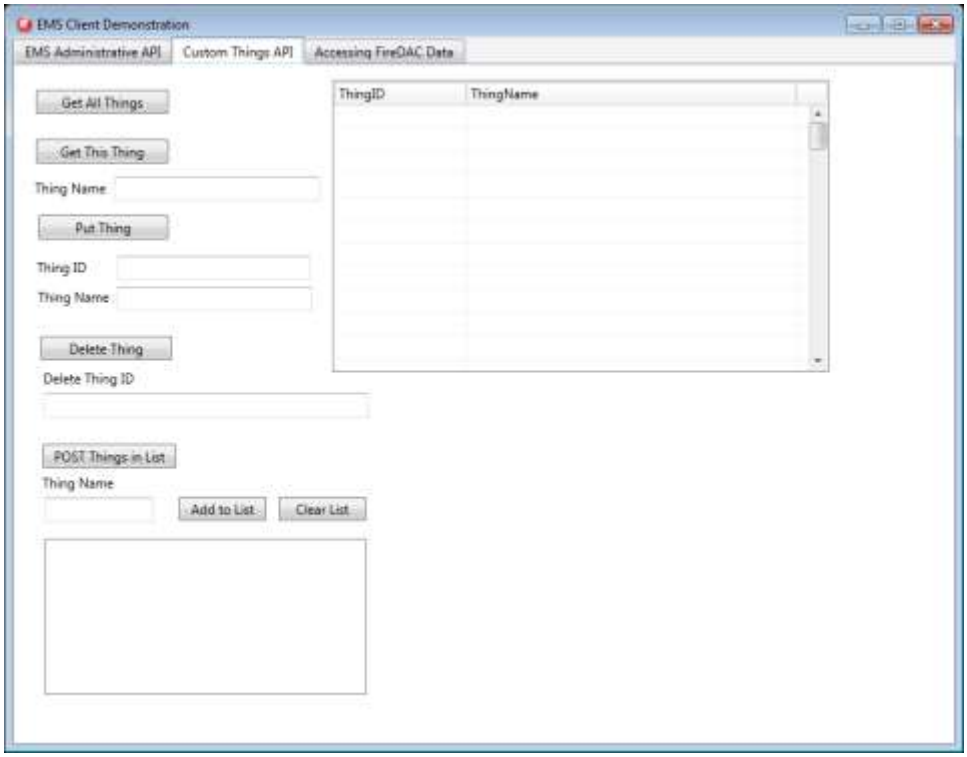
Some browsers are kind enough to actually display the error message returned in the exception. For example, this is how the response to this same request appears in Google Chrome:




One thing is for sure. This Things resource is available and its endpoints are exposed. However, we are not going to be able to create a new Thing from a browser (without

involving some JavaScript), since that requires either a PUT or POST request. So, let's turn our attention to the EMSDemoClient application, which has implemented calls to all of the endpoints of the Things resource.

The following is the Custom Things API page of the EMSDemoClient project:




Since we don't have any Things, let's start by creating one. In order to do this, we need a component that can talk to a custom endpoint, which must be a component that can issue HTTP requests.

RAD Studio has a lot of components that can satisfy this need. We could use Internet Direct (Indy) components, such as TIdTCPClient or TIdHTTPClient. Or, we could use a RESTRequest from the new RAD Studio REST client library. In this first example, I will demonstrate this call with a BackendEndpoint component, one specifically designed to make access to EMS resources easy.

Here is the code located on the button labeled Put Thing:

```
procedure TForm1.PutButtonClick(Sender: TObject);
begin
  dm.BackendEndpointPut.Params[0].Value := '{"request":["' + IDEdit.Text
+ '", "' +

NameEdit.Text + '"]}';
  dm.BackendEndpointPut.ExecuteAsync;
end;
```

BackendEndpointPut is a BackendEndpoint component, and it is configured to use the EMSProvider that all Backend components on this project use, and that component is pointed to the EMS server. Other properties that are set include Method, set to rmPut, Resource, set to Things, ResourceSuffix, set to {Item}, and Reponse, set to GetReponse, a TRESTResponse object. These properties instruct the BackendEndpoint component to make a PUT request of the Things resource, passing a single argument in the HTTP message body. When the response is received, it will be assigned to the TRESTResponse object named GetResponse, from which the response data can be read.

The code in the preceding event handler creates a JSON object that passes an array of two strings in its request property. The first string in this array is the Thing ID, and the second is the Thing Name. I could have used some other format, and that would be fine, just so long as the client and server agree about the format of the information they are passing back and forth.

After assigning a value to the parameter, the ExecuteAsync method is called, which executes the request asynchronously. ExecuteAsync can take a parameter that identifies a method that will be executed when the response has been received, but that wasn't necessary in this case.

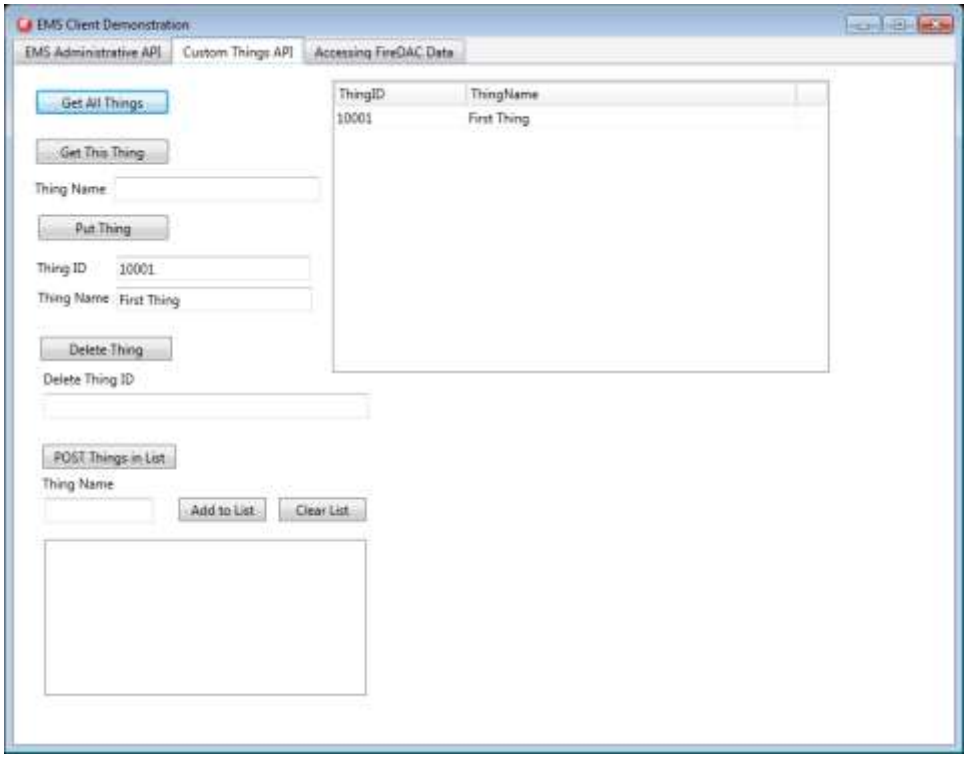
Before we execute the PUT request, let's consider another endpoint, Get, which gets a list of all Things. The following is the code that appears in the OnClick event handler of the button labeled Get All Things. I've also included the implementations of the GetThingsDone and LoadThingsGrid methods, since they are invoked when the Get invocation is successful:

```
procedure TForm1.GetThingsBtnClick(Sender: TObject);
begin
  dm.BackendEndpointGet.ExecuteAsync(GetThingsDone);
end;

procedure TForm1.GetThingsDone;
begin
  LoadThingsGrid(dm.GetResponse.Content);
end;

procedure TForm1.LoadThingsGrid(JSONString: string);
var
  jo: TJSONObject;
  ja, ja1: TJSONArray;
  jv: TJSONValue;
begin
  FDMemTable1.Close;
  FDMEmTable1.Active := True;
```

```
  jo := TJSONObject.ParseJSONValue(JSONString) as TJSONObject;
  ja := jo.GetValue('response') as TJSONArray;
  for jv in ja do
  begin
    ja1 := jv as TJSONArray;
    FDMemTable1.InsertRecord([ja1.Get(0).Value, ja1.Get(1).Value]);
  end;
  jo.Free;
end;
```

BackendEndpointGet is a BackendEndpoint, and it is configured similar to BackendEndpointPut. Provider points to EMSProvider, Method is set to rmGet, Resource is set to Things, Response is set to GetReponse, and ResourceSuffix is an empty string, since this endpoint requires no parameters. When we call this BackendEndpoint's ExecuteAsync method we do want to pass a method to be invoked when the response is received, since we need to handle the response. This method, called GetThingsDone, calls LoadThingsGrid, passing to that method the Content property of the response object. As you can see from the implementation of LoadThingsGrid, this method creates a JSON object from the string contents of the RESTResponse, after which it loads the grid with the data about Things.

In the following figure, the Put Thing button has been clicked, after which the Get All Things button has been clicked. The one Thing that was inserted into the database is now being displayed in the provided grid:



Before continuing, I want to emphasis that the BackendEndpoint components make working with EMS servers easy, but there are alternatives. For example, the EMSDemoClient includes several components of the REST client library, and they are just as capable of calling EMS REST endpoints as the Backend components. The REST client library components just require a little more code.

The Custom Things API page demonstrates how to use Backend component to access all of the endpoints of the Things resource. Some of the event handlers also show code samples that call these endpoints using the REST client library. For more information, please refer to the source files in the EMSDemoClient application.

## Creating Additional Custom Endpoints

You are not limited to the endpoints created by the EMS Package Wizard, and you are free to modify those created by that Wizard. For example, you might initially tell the EMS Package Wizard to create GET and POST endpoints for your resource. If you later decide you need a PUT instead the POST, you can delete the Post method and add a PutItem method that has the same parameters as the other methods. You will also need to ensure that you define a valid ResourceSuffix attribute for any endpoints you add.

The rules for declaring new, custom endpoints for a particular resource are pretty straightforward. Each endpoint procedure name, which is case insensitive, must begin with one of the following prefixes: Get, Patch, Post, Put, or Delete. In addition, each endpoint must use the same parameters as you see in those generated by the EMS Package Wizard. Specifically, each method must have three parameters of type TEndpointContext, TEndpointRequest, and TEndpointResponse, respectively.

The HTTP method associated with a particular endpoint, which indicates the method by which the REST client invokes the method and passes data, is defined by a prefix to the endpoint name. For example, if you have an endpoint named Customer, a method named GetCustomer is invoked using an HTTP GET, and a method named DeleteCustomer is invoked using an HTTP DELETE.

ResourceSuffix attributes define the endpoint name and any parameters, each of which represent a segment of the URL. ResouceSuffix attributes that appear in clear text (no curly braces) are literal URL segments, and those that appear in curly braces are parameters.

Let's look at a couple of examples that should make this clear. Consider the somewhat banal ReverseString method that you might have seen generated in DataSnap examples. The following declaration will add this method to the Demo resource you

created earlier. Notice that the text ReverseString in the ResourceSuffice attribute is not enclosed in curly braces, meaning that it is a segment of the URL (regardless of the method type implied by the Get part of the actual method name):

```
[ResourceSuffix('ReverseString/{item}')]
procedure GetReverseString(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
```

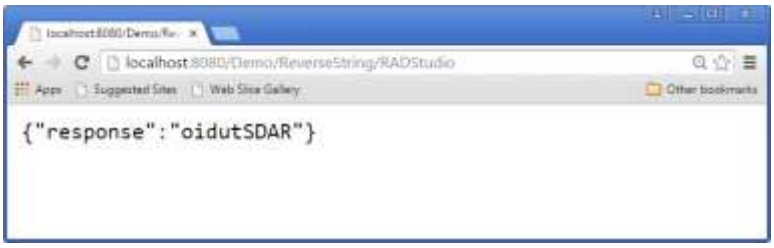
Here is how this method might be implemented:

```
procedure TDemoResource1.GetReverseString(const AContext:
TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  LItem: string;
begin
  LItem := ARequest.Params.Values['item'];
  LItem := ReverseString(LItem);
  AResponse.Body.SetValue(
                TJSONObject.Create(TJSONPair.Create('response', LItem)),
True);
end;
```

The URL you use to invoke this method in a browser should look something like this:

```
http://localhost:8080/Demo/ReverseString/RADStudio
```

And here is how this GET method looks after being invoked through a browser:




It is even possible to have more than one URL segment in the ResourceSuffix, which can be a useful tool for organizing your endpoints within a particular resource (though it is just as valid, and in some cases, more so, to organize endpoints by resource, as you can have many resources in a single EMS package).

Consider this method declaration:

```
ResourceSuffix('DateTime/ServerTimestamp')]
procedure GetDateTimeServerTimestamp(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
```

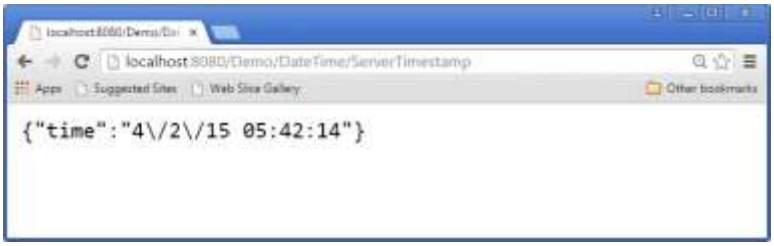
And this implementation:

```
procedure TDemoResource1.GetDateTimeServerTimestamp(const AContext:
TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
    AResponse.Body.SetValue(
      TJSONObject.Create(TJSONPair.Create('time',
                         FormatDateTime('m/d/yy hh:mm:ss', now))),True)
end;
```

Here is how this endpoint would be referenced in a browser:

```
http://localhost:8080/Demo/DateTime/ServerTimestamp
```

And here is how the response might appear:




Just as you can have more than one URL segment, you can have more than one parameter. For example, consider this method declaration:

```
[ResourceSuffix('Concat/{str1}/{str2}')]
procedure GetConcat(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
```

And implemented this way:

```
procedure TDemoResource1.GetConcat(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  AResponse.Body.SetValue(
    TJSONObject.Create(
      TJSONPair.Create('response',
        ARequest.Params.Values['str1'] +
        ARequest.Params.Values['str2'])) , True);
end;
```

Here is the invocation:

```
http://localhost:8080/Demo/Concat/EMS/Server
```

And here is this endpoint in action:


The only thing you have to watch out for is to ensure that the EMS server can distinguish between the various endpoints in your resource. If the EMS server reports a conflict between two or more of your endpoints, you may have to move one or more of them to another resource, or add an additional URL segment to the resource suffix.

The sample projects included with this white paper include EMSDemoPkg. This package defines custom EMS resources that can be consumed by the EMSDemoClient application. You can get some additional ideas about creating custom resources and endpoints by examining the various classes in this package, and the corresponding code in the client application that consumes these resources.

# Adding Additional Resources to an Existing Package

As mentioned in the preceding section, when you have a lot of endpoints in an EMS resource, you might benefit from using multi-segment URL endpoints for organizational purposes. Rather than taking that approach, however, you might want to consider adding additional resources to an existing EMS package. In that way you can organize your endpoints by resource. Or, you can use multiple resources and multi-segment URL endpoints, if that suits your needs.

EMS Module

Adding additional resources to an existing package is simple. With your custom EMS package selected in RAD Studio's Project Manager, select File | New | Other from RAD Studio's main menu, and then double-click on the EMS Module Wizard from the EMS tab of the Object Repository:

The EMS Module Wizard consists of the last two pages of the EMS Project Wizard, the Resource page and the Endpoints page. Use the Resource page to provide a name for your new resource, and select whether you want this resource to be based on a data module or a TObject. Use the Endpoints page to select which endpoints you want the EMS Module Wizard to generate into your new resource class. When you click Finish,

the EMS Module Wizard adds a new unit to your EMS package. This unit contains your new resource definition, stubs for the requested endpoints, and a Register procedure that registers your new resource.

There is an alternative way to create new resources in an existing EMS package. You can simply create one or more additional class declarations within your existing units, decorating each class with a correctly formed ResourceName attribute, and adding additional calls to RegisterResource to the existing Register procedure, one for each new resource class that you have created. The resulting resource is really no different than the one that would be generated by the EMS Module Wizard, other you would be maintaining more than one resource in each unit, and it requires a bit more typing.

# EMS Custom Packages and FireDAC

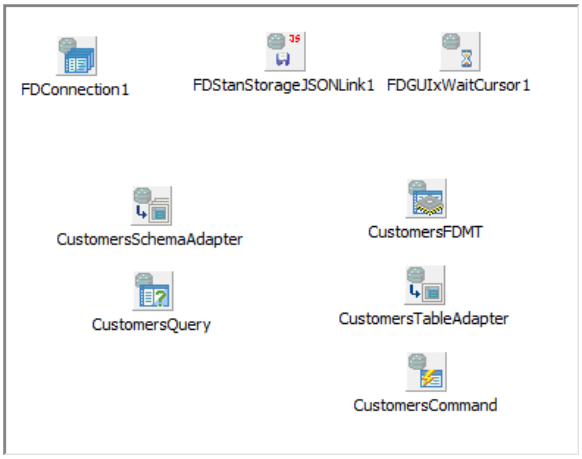
You've learned that you can communicate with your EMS server endpoints using any framework that supports HTTP. You've also learned that there are a number of specialized components in RAD Studio that are specifically designed to simplify your interaction with EMS servers. These include EMSProvider, BackendEndpoints, BackendUsers, BackendGroups, and BackendAuth.

There is one more component worth mentioning, EMSFireDACClient. Unlike the Backend components, which are general purpose client components for communicating with an EMS server, the EMSFireDACClient has a very specific application. It facilitates the communication between a FireDAC FDSchemaAdapter on an EMS server and another in a client. When used in conjunction with properly configured FireDAC components, the EMSFireDACClient can retrieve data from the EMS server with as little as one line of code. Posting the data back to the server again requires only one line of code.

EMSFireDACClient can only be used in EMS client applications written in RAD Studio. Nonetheless, its ease of use, once the necessary components have been properly configured, can make working with remote data nearly effortless, at least from the client-side of things.

In this section, you are going to learn how to transfer a FireDAC FDMemTable between an EMS server and an EMS client using two different mechanisms. One of these employs FDMemTables on the client and server, and does not require the use of FDSchemaAdapters. This technique employs a BackendEndpoint on the client to enable the communication. The second, which does make use of FDSchemaAdapters, employs an EMSFireDACClient on the client.

The EMSDemoPkg package includes a resource named FireDACDemo, and it is implemented as a data module whose design surface is shown in the following figure:




The two components that are exposed though this resource's endpoints are the FDSchemaAdapter, named CustomersSchemaAdapter, and the FDMemTable, named CustomersFDMT. The FDQuery, named CustomersQuery, is connected to the FDConnection, which is configured to use the dbdemos.gdb database that ships with RAD Studio. The query holds the following SQL statement, which matches the SQL statement also defined for the FDCommand, named CustomersCommand:

```
SELECT * FROM Customer
```

CustomerQuery is connected to CustomersSchemaAdapter through its SchemaAdapter property. CustomersFDMT is connected to CustomersTableAdapter, an FDTableAdapter, through its Adapter property. CustomersTableAdapter is connected to CustomersCommand through its SelectCommand property.

Let's now look at the definition of the FireDACDemo resource:

```
type
  [ResourceName('FireDACDemo')]
  TFireDACDemoResource = class(TDataModule)
    FDConnection1: TFDConnection;
    FDGUIxWaitCursor1: TFDGUIxWaitCursor;
    FDStanStorageJSONLink1: TFDStanStorageJSONLink;
    CustomersSchemaAdapter: TFDSchemaAdapter;
    CustomersQuery: TFDQuery;
    CustomersFDMT: TFDMemTable;
    CustomersTableAdapter: TFDTableAdapter;
    CustomersCommand: TFDCommand;
  private
  public
    constructor Create(AOwner: TComponent); override;
  published
```

```
    [ResourceSuffix('Customers')]
    procedure GetCustomers(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('CustomersFDMT')]
    procedure GetCustomersFDMT(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('Customer')]
    procedure GetCustomer(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('CustomerFDMT')]
    procedure GetCustomerFDMT(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('Customers')]
    prbocedure PostCustomers(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
    [ResourceSuffix('CustomersFDMT')]
    procedure PostCustomersFDMT(const AContext: TEndpointContext;
      const ARequest: TEndpointRequest; const AResponse:
TEndpointResponse);
  end;
```

There are six REST endpoints exposed by this resource. Four of these are GET endpoints, where two endpoints expose the FDSchemaAdapter and two expose the FDMemTable. For each component, one of these endpoints returns the entire resource, and the other endpoint reads a value passed by the client, permitting a single record to be queried and returned to the client. The final two endpoints are POST endpoints, and they receive and apply the updates made by the client application.

Here are the implementations of the two GET endpoints. One gets all customers using the FDSchemaAdapter, and the other gets only one record, using the FDMemTable:

```
procedure TFireDACDemoResource.GetCustomers(const AContext:
TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  ms: TMemoryStream;
begin
  ms := TMemoryStream.Create;
  try
    CustomersQuery.SQL.Text := 'SELECT * FROM Customer';
    CustomersQuery.Open;
    CustomersSchemaAdapter.SaveToStream(ms, TFDStorageFormat.sfJSON);
    AResponse.Body.SetStream(ms, 'application/json', True);
  except
    ms.Free;
  end;
end;
```

```
procedure TFireDACDemoResource.GetCustomerFDMT(const AContext:
TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
var
  ms: TMemoryStream;
begin
  ms := TMemoryStream.Create;
  try
    CustomersFDMT.CommandText := 'SELECT * FROM Customer WHERE CustNo =
:cust';
    CustomersFDMT.ParamByName('cust').AsString :=
ARequest.Params.Values['data'];
    CustomersFDMT.Open;
    CustomersFDMT.SaveToStream(ms, TFDStorageFormat.sfJSON);
    AResponse.Body.SetStream(ms, 'application/json', True);
  except
    ms.Free;
  end;
end;
```

In both cases, the data from the corresponding component is written to a stream, after which that stream is send in the response to the client.

The POST requests perform the somewhat similar operation, though in the reverse direction. Specifically, data is read from a stream in the request and assigned to the corresponding component, after which ApplyUpdates is called. Here are the two POST implementations:

```
procedure TFireDACDemoResource.PostCustomers(
  const AContext: TEndpointContext; const ARequest: TEndpointRequest;
  const AResponse: TEndpointResponse);
var
  s: TStream;
begin
  if not ARequest.Body.TryGetStream(s) then
    AResponse.RaiseBadRequest('no stream');
  CustomersQuery.SchemaAdapter := nil;
  CustomersSchemaAdapter.LoadFromStream(s, TFDStorageFormat.sfJSON);
  CustomersSchemaAdapter.ApplyUpdates;
end;

procedure TFireDACDemoResource.PostCustomersFDMT(
  const AContext: TEndpointContext; const ARequest: TEndpointRequest;
  const AResponse: TEndpointResponse);
var
  s: TStream;
begin
  if not ARequest.Body.TryGetStream(s) then
    AResponse.RaiseBadRequest('no stream');
  CustomersFDMT.LoadFromStream(s, TFDStorageFormat.sfJSON);
  CustomersFDMT.ApplyUpdates;
end;
```

Let's now take a look at the client for this resource. I am going to focus first on the traditional way that this stream is requested and displayed. Here is the code associated with the Get CustomerFDMT:

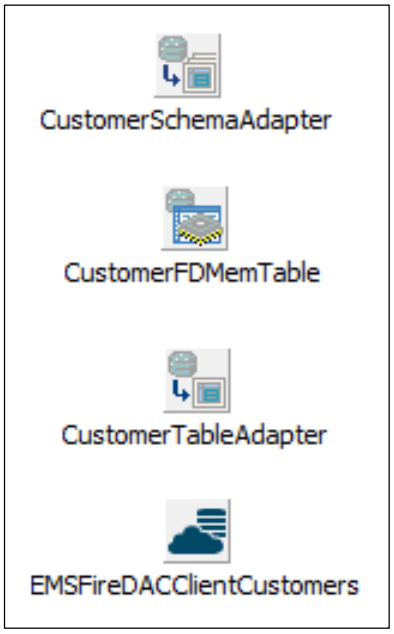
```
procedure TForm1.GetCustomerFDMTBtnClick(Sender: TObject);
var
  s: TStringStream;
begin
  dm.BackendEndpointCustomersFDMT.Method := rmGET;
  dm.BackendEndpointCustomersFDMT.Params.Clear;
  if CustNoFDMTEdit.Text = '' then
  begin
    dm.BackendEndpointCustomersFDMT.Resource :=
'FireDACDemo/CustomersFDMT';
    dm.BackendEndpointCustomersFDMT.Execute;
    s := TStringStream.Create(dm.GetResponse.Content);
    try
      dm.FDMTCustomerFDMemTable.LoadFromStream(s,
TFDStorageFormat.sfJSON);
    finally
      s.Free;
    end;
  end
  else
  begin

dm.BackendEndpointCustomersFDMT.AddParameter('data',CustNoFDMTEdit.Text);
    dm.BackendEndpointCustomersFDMT.Resource :=
'FireDACDemo/CustomerFDMT';
    dm.BackendEndpointCustomersFDMT.Execute;
    s := TStringStream.Create(dm.GetResponse.Content);
    try
      dm.FDMTCustomerFDMemTable.LoadFromStream(s,
TFDStorageFormat.sfJSON);
    finally
      s.Free;
    end;
  end;
  dm.FDMTCustomerFDMemTable.CachedUpdates := True;
end;
```

The BackendEndpoint used in this code is wired to an EMSProvider and a BackendAuth component (in case we implement authorization). Otherwise, the other properties are set in this code. The TEdit named CustNoFDMTEdit is provided in the user interface. If the user has entered a customer number, the CustomerFDMT endpoint is requested, passing the value entered by the user in a parameter. Otherwise, CustomersFDMT is called. When the data is received it is loaded into the FDMemTable and cached updates is turned on.

Sending this data back requires a similar set of operations. Here is the code associated with the button that a user clicks to post changes back to the server. Note that it does not matter if a single record was originally received, or the entire customer table.

```
procedure TForm1.PostCustomerFDMTBtnClick(Sender: TObject);
var
  s: TMemoryStream;
begin
  s := TMemoryStream.Create;
  dm.FDMTCustomerFDMemTable.SaveToStream(s, TFDStorageFormat.sfJSON);
  s.Position := 0;
  dm.BackendEndpointCustomersFDMT.Resource := 'CustomersFDMT';
  dm.BackendEndpointCustomersFDMT.Method := rmPOST;
  dm.BackendEndpointCustomersFDMT.AddBody(s,
                                    TRESTContentType.ctAPPLICATION_JSON);
  dm.BackendEndpointCustomersFDMT.ExecuteAsync();
end;
```

Let's now turn our attention to the EMSFireDACClient component. This component is designed to transmit an FDSchemaAdapter between the client and the EMS server. It also requires a set of properly configured components on the client. These are shown here, and can be found in the data module of the EMSDemoClient project.




CustomerTableAdapter is an FDTableAdapter that points to CustomerSchemaAdapter in its SchemaAdapter property. Also, the DatSTableName property of CustomerTableAdapter is set to CustomerQuery, the name of the FDQuery object on the FireDACDemo data module from which the server-side FDSchemaAdapter retrieved its data.

CustomerFDMemTable is an FDMemTable, and it will hold the data on the client, and cache the users updates made to that data. CustomerFDMemTable refers to CustomerTableAdapter in its Adapter property.

EMSFireDACClientCustomers is an EMSFireDACClient, and its Provider property points to an EMSProvider and its Auth property points to a BackendAuth component. These properties, as well as Resource, are the same as those for a BackendEndpoint component. The one unique property of EMSFireDACClient is SchemaAdapter, and in this case it has been assigned CustomerSchemaAdapter.

Now that everything is wired up properly, using the EMSFireDACClient is simple. Here is the code associated with the GET endpoints. Like that for the FDMemTable, a TEdit is

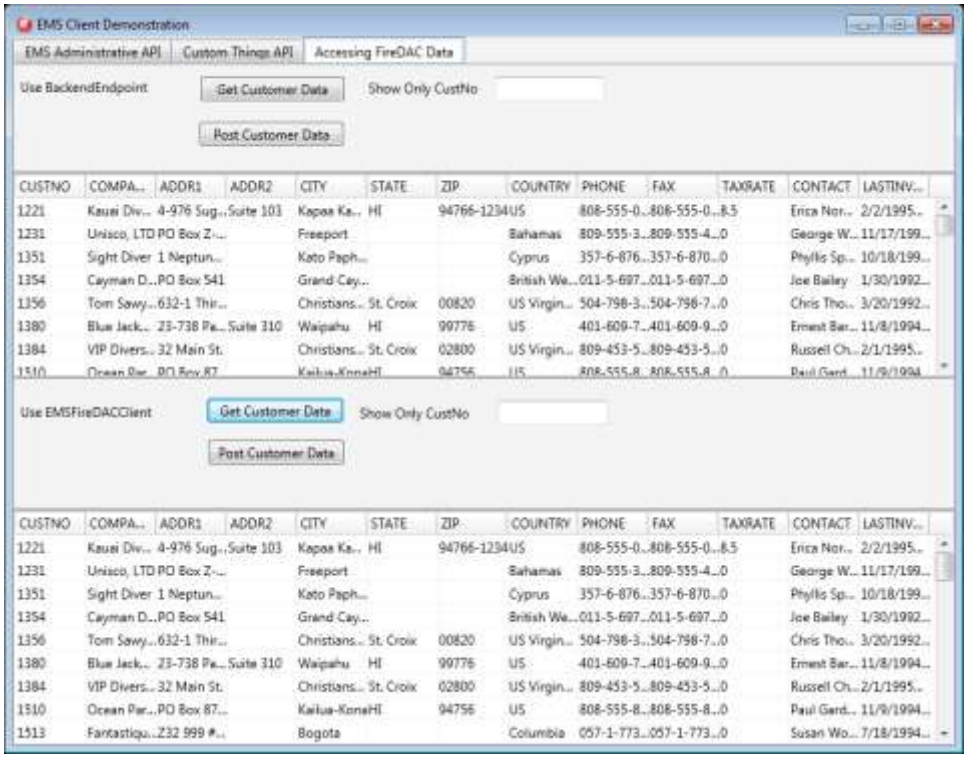
used to permit a user to select just one or all of the Customer table records, and a parameter is used to pass the identifier of a single record. Otherwise, the code is basically the same as that for retrieving the FDMemTable:

```
procedure TForm1.GetCustomerSchemaBtnClick(Sender: TObject);
begin
  dm.EMSFireDACClientCustomers.GetEndpoint.Params.Clear;
  if CustNoSchemaBtn.Text = '' then
  begin
    dm.EMSFireDACClientCustomers.Resource := 'FireDACDemo/Customers';
    dm.EMSFireDACClientCustomers.GetEndpoint.Params.Clear;
    dm.EMSFireDACClientCustomers.GetData;
  end
  else
  begin
    dm.EMSFireDACClientCustomers.Resource := 'FireDACDemo/Customer';
    dm.EMSFireDACClientCustomers.GetEndpoint.AddParameter('data',
                                          CustNoSchemaBtn.Text);
    dm.EMSFireDACClientCustomers.GetData;
  end;
  dm.CustomerFDMemTable.CachedUpdates := True;
end;
```

Sending the data back using the associated POST endpoint is even simpler, as shown here:

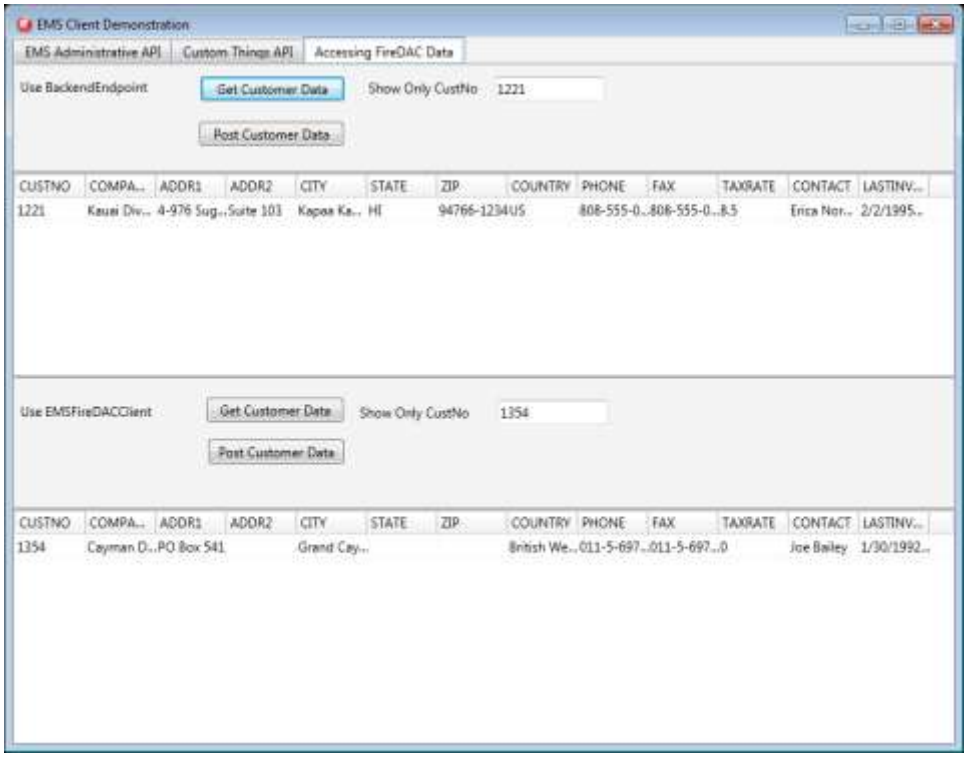
```
procedure TForm1.PostCustomerSchemaBtnClick(Sender: TObject);
begin
  dm.EMSFireDACClientCustomers.Resource := 'Customers';
  dm.EMSFireDACClientCustomers.GetEndpoint.Params.Clear;
  dm.EMSFireDACClientCustomers.PostUpdates;
end;
```

The following figure shows the EMSDemoClient project. In this figure, two complete copies of the Customer table have been retrieved from the EMS server:




| CUSTNO | COMPA... | ADDR1 | ADDR2 | CITY | STATE | ZIP | COUNTRY | PHONE | FAX | TAXRATE | CONTACT | LASTINV... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1221 | Kauai Div... | 4-976 Sug... | Suite 103 | Kapaa Ka... | HI | 94766-1234 | US | 808-555-0... | 808-555-0... | 8.5 | Erica Nor... | 2/2/1995... |
| 1231 | Unisco, LTD | PO Box Z-... | | Freeport | | | Bahamas | 809-555-3... | 809-555-4... | 0 | George W... | 11/17/199... |
| 1351 | Sight Diver | 1 Neptun... | | Kato Paph... | | | Cyprus | 357-6-876... | 357-6-870... | 0 | Phyllis Sp... | 10/18/199... |
| 1354 | Cayman D... | PO Box 541 | | Grand Cay... | | | British We... | 011-5-697... | 011-5-697... | 0 | Joe Bailey | 1/30/1992... |
| 1356 | Tom Sawy... | 632-1 Thir... | | Christians... | St. Croix | 00820 | US Virgin... | 504-798-3... | 504-798-7... | 0 | Chris Tho... | 3/20/1992... |
| 1380 | Blue Jack... | 23-738 Pa... | Suite 310 | Waipahu | HI | 99776 | US | 401-609-7... | 401-609-9... | 0 | Ernest Bar... | 11/8/1994... |
| 1384 | VIP Divers... | 32 Main St. | | Christians... | St. Croix | 02800 | US Virgin... | 809-453-5... | 809-453-5... | 0 | Russell Ch... | 2/1/1995... |
| 1510 | Ocean Par... | PO Box 87... | | Kailua-Kona | HI | 94756 | US | 808-555-8... | 808-555-8... | 0 | Paul Gard... | 11/9/1994... |
| 1513 | Fantastiqu... | 232 999 #... | | Bogota | | | Columbia | 057-1-773... | 057-1-773... | 0 | Susan Wo... | 7/18/1994... |

Here is this same project, but in this case only a single record of the Customer table has been requested by the client:




| CUSTNO | COMPA... | ADDR1 | ADDR2 | CITY | STATE | ZIP | COUNTRY | PHONE | FAX | TAXRATE | CONTACT | LASTINV... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1221 | Kauai Div... | 4-976 Sug... | Suite 103 | Kapaa Ka... | HI | 94766-1234 | US | 808-555-0... | 808-555-0... | 8.5 | Erica Nor... | 2/2/1995... |

| CUSTNO | COMPA... | ADDR1 | ADDR2 | CITY | STATE | ZIP | COUNTRY | PHONE | FAX | TAXRATE | CONTACT | LASTINV... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1354 | Cayman D... | PO Box 541 | | Grand Cay... | | | British We... | 011-5-697... | 011-5-697... | 0 | Joe Bailey | 1/30/1992... |

Although you cannot tell from this figure, if you make changes to any of the data that appears on this page, and then click the associated Post Customer Data button, that data will be returned to the EMS server where the corresponding POST endpoint will attempt to apply that data to the underlying InterBase database.

# Deploying EMS as an ISAPI Server

When you purchase an EMS license you will receive an installer that contains, among other things, ISAPI DLL versions of both the EMS server and EMS console. These permit you to run EMS under IIS, a highly available web server service that provides for additional security.

There is one very big advantage, from my perspective, to running EMS under IIS. When using the EMS ISAPI DLL, you can use web pages served from the same IIS server to invoke the endpoints of your EMS server. I'm not saying that you should do this, but it does provide secure access to your organization's data and functionality from any browser, without requiring the installation of an application.

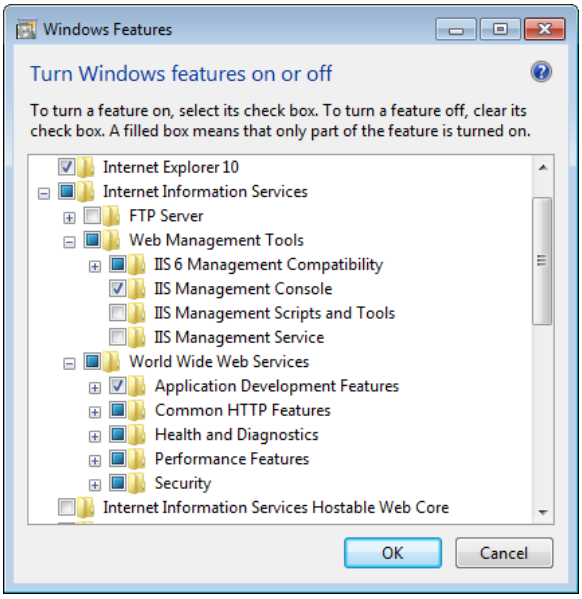
You cannot, on the other hand, access the features of the EMS development service using a web page served from IIS. The reason is that IIS and the EMS development server are listening on separate ports, and EMS prevents access from the outside request due to its restrictions on cross-domain posting.

In this section, I am going to describe how to configure your server to run ISAPI DLLs under IIS. After that, I will discuss the minor adjustments that you need to make to your URLs in order to access EMS through IIS.

## Preparing to Run an ISAPI DLL

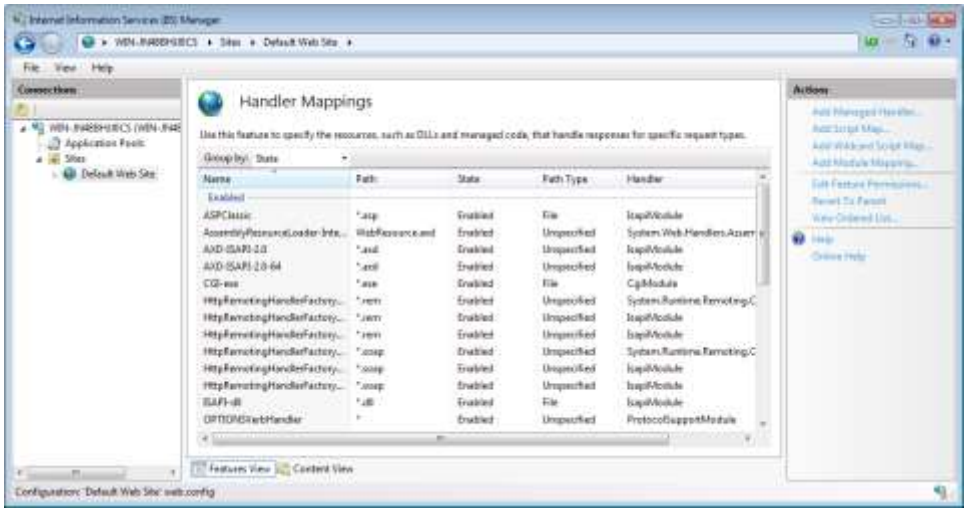
The following steps demonstrate how to install and configure Microsoft's Internet Information Services version 7.

To begin with, you want to install IIS and the IIS Manager. To do this, open the Programs and Features applet from the control panel, and then click on the Turn Windows features on and off link to display the Windows Features dialog box, shown in the following figure. In this figure I have expanded the Internet Information Services node in order to display the options that I have selected for my development environment.

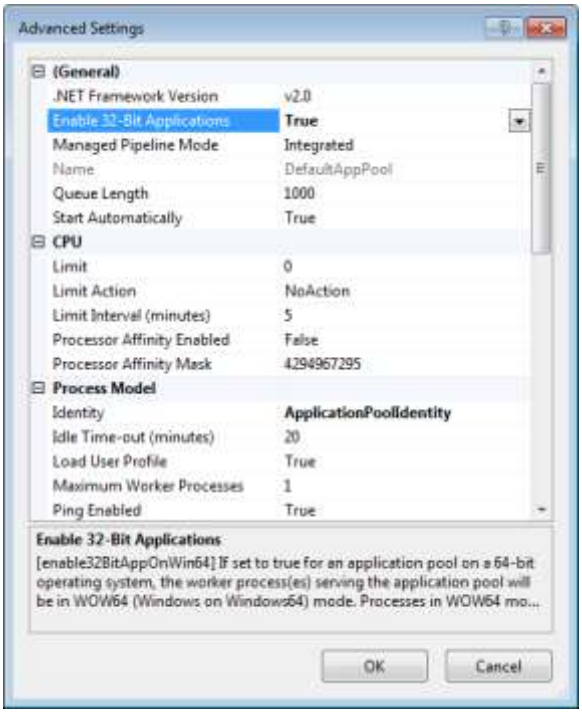



When you are done selecting the Windows features to install, click OK.

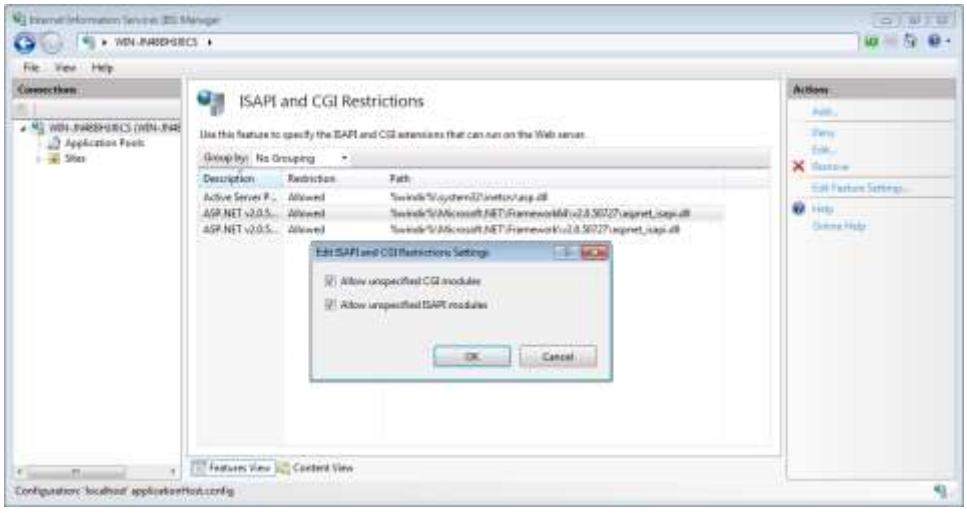
Next, you must enable the execution of ISAPI or CGI applications. To so this, open the Internet Information Services Manager console, which you will find in the Administrative Tools applet in the control panel. Select the Default Web site and open Handler Mappings. You can either enable the ISAPI or CGI handlers that are present by default (and they are disabled, by default), or you can create a specific handler for your application, enabling it at the time you create it. In the following figure, I have enabled both CGI and ISAPI applications.




If you are running 64-bit Windows, and you are installing a 32-bit ISAPI DLL, you must also enable 32-bit applications on the application pool on which your ISAPI DLL will run. In my case, it is the DefaultAppPool. The following figure shows the Advanced Settings dialog box for the DefaultAppPool:




There are other steps you can take, but these are not always necessary. For example, you can enable "Allow unspecified CGI modules" and/or "Allow unspecified ISAPI modules" for the entire web site. To do this, select the top-level node in the Connections pane of the Internet Information Services Manager console. Next, open ISAPI and CGI Restrictions. Right-click in the ISAPI and CGI Restrictions pane and select Edit Feature Settings. At a minimum, place a checkmark next to Allow unspecified ISAPI modules, as shown in the following figure. There is some security risk in doing this, since it does not apply to your specific applications, but sometimes this step is necessary.

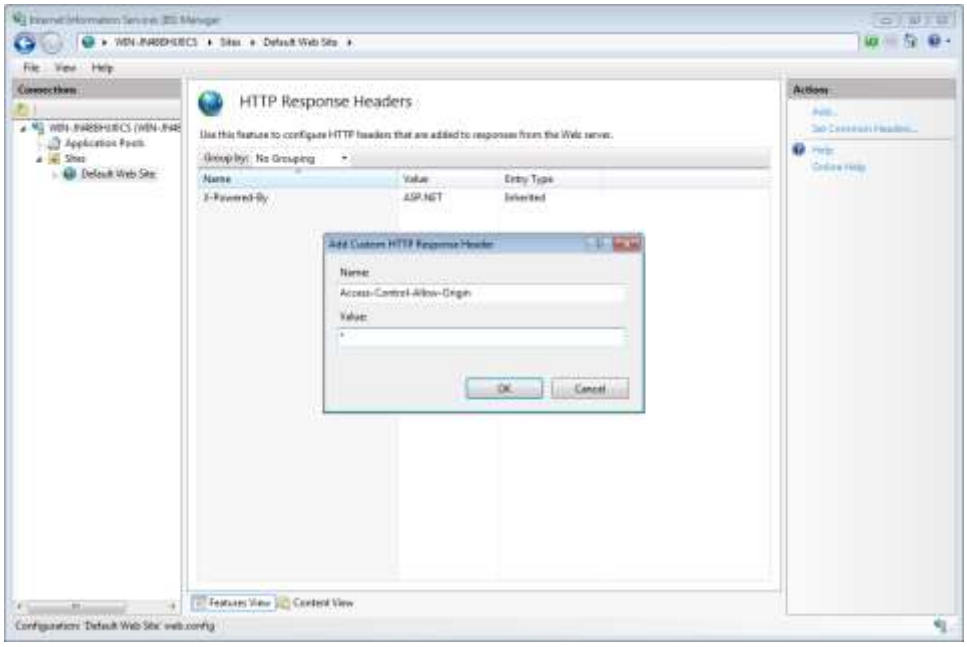


The final step, if you plan on calling into your EMS endpoints using JavaScript executing on a web page served by IIS, is to allow cross-domain posting (also known as Cross-Origin Resource Sharing, or CORS for short). You will need to use this setting as well if you plan on using the ISAPI DLL version of the EMS console, which must also access the EMS server from web pages served by IIS.

Begin by selecting Default Web Site under the Sites node. If you want to have even tighter control, expand Default Web Site and select the specific directory in which the HTML pages that access your EMS service will be served from (this setting will apply to those directories and all of their subdirectories).

Next, double-click the HTTP Response Headers icon in the IIS pane. Next, click the Add button in the Actions pane to add a custom HTTP response header.

Set Name to Access-Control-Allow-Origin, and Value to *, as shown in the following figure. Value can also be set to a specific domain or IP address and port, if you are concerned about other servers making cross-domain posts.




These settings will get you started, and are provided to help you run the EMS server on your development machine. If you are configuring IIS for EMS deployment on a server that is exposed to the Internet, please seek advice on properly securing your IIS installation.

# Installing the EMS ISAPI Server and Console

At the time of this writing the EMS deployment server setup program copies 32-bit versions of the EMS server and console ISAPI DLLs to the bin directory under the RAD Studio installation directory. I am going to assume that this is still the case.

Create a new folder under c:\inetpub\wwwroot on your server (or your development machine, if that's where you want to test EMS). I have called mine EMSServer.

Copy the EMSServer.dll and EMSConsole.DLL files from the bin directory in which they were installed to your newly created folder.

Next, copy the emsserver.ini file that you've been using to this same folder. You should now use this copy of the ini file to make changes to your EMS server and EMS console configuration. Note that since this configuration file is now under c:\inetpub\wwwroot, you will need to edit this file using an editor with Administrative privileges, otherwise you will not be able to save your changes. I run RAD Studio as an Administrator, and I add the emsserver.ini file to my custom EMS package project, making it both readily available and easy to edit and save.

There is one final step. You need to instruct the EMS server ISAPI DLL to load any custom EMS packages each time it is loaded by IIS. Open the copy of emsserver.ini that resides in the same directory as your EMSServer.dll. Find the Server.Packages section in the configuration file, and add the fully qualified path to the runtime package as an entry in this section. You will need to do this for each custom package that you have created.

My demonstration package is named EMSDemoPkg.bpl, and my Server.Packages section looks like the following listing. Note that I have also added the DemoEMS package that we created using the EMS Package Wizard:

```
[Server.Packages]
;# This section is for extension packages.
;# Extension packages are used to register custom resource endpoints
;c:\mypackages\basicextensions.bpl=mypackage description
C:\Users\Public\Documents\Embarcadero\Studio\15.0\Bpl\EMSDemonPkg.bpl=Dem
o package
C:\Users\Public\Documents\Embarcadero\Studio\15.0\Bpl\DemoEMSPackage.bpl=
Created Demo
```

(Note that the last two lines of this entry may have wrapped, due to the format of this paper. If that is the case, note that these entries must appear must not be wrapped in the ini file.)

The EMS console ISAPI DLL, like its developer server counterpart, relies on a collection of files to display a user interface within a browser. You must take additional steps to make those files available to the EMS console ISAPI DLL. First, you must place those files in a directory to which cross-domain posting has been explicitly permitted (as described earlier in this section). You must also identify that location in the EMS configuration file.

You will find the EMS console web files in a directory named webresources located under the en/ems folder, which is located in the Object Repository folder of your RAD Studio installation. In RAD Studio XE7, the webresources folder can be found in the following location:

```
C:\Program Files (x86)\Embarcadero\Studio\15.0\ObjRepos\en\EMS
```

Copy this folder, in its entirety, to a location under wwwroot. In my installation, I placed a copy of this folder in the same folder into which I copied the EMS ISAPI DLLs and the EMS configuration file, which in my case is the following directory:

```
c:\inetpub\wwwroot\emsserver
```

Next, update the Console.Paths.ISAPI section of your EMS configure file, setting the ResourcesFiles key to the path where your webresources folder is located. My entry looks like the following:

```
[Console.Paths.ISAPI]
ResourcesFiles = C:\inetpub\wwwroot\emsserver\
```

## Using Your EMS Server ISAPI DLL

There are very few changes that you need to make in order to invoke the ISAPI DLL of your EMS server, once you've properly installed and configured it. For one, so long as you have IIS listening on the default HTTP and HTTPS ports, you will no longer have to include a port segment in your URL. You will, however, have to reference the virtual directory into which you've copied the server, and the server as well, unless you define the server as one of the default resources in this directory.

On my system, I have copied EMSServer.dll to c:\inetpub\wwwroot\emsserver. As a result, I can invoke the Version endpoint using the following call:

```
http://localhost/emsserver/emsserver.dll/Version
```

Here is how this invocation looks in Google Chrome:


Let now try one of the more complex, multi-segment URLs. Here is now we invoke the ServerTimesamp endpoint using the ISAPI DLL:

```
http://localhost/emsserver/emsserver.dll/Demo/DateTime/ServerTimestamp
```

Here is the response:


From RAD Studio's EMS components, things are a little different as well. For example, when you configure the EMSProvider to run against the EMS development server, you can leave the URLBasePath property blank. However, we now have a base URL path, which is the virtual directory reference to the DLL, and it looks like this:

```
emsserver/emsserver.dll
```

and, of course, instead of using a URLPort value of 8080, you will use 80 (for HTTP) or 443 (for HTTPS).

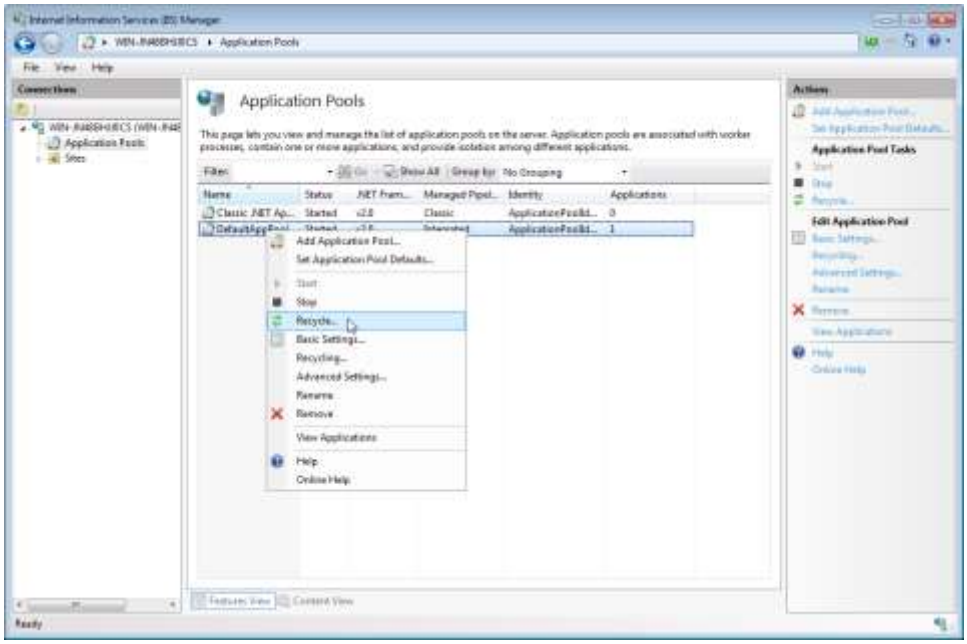
Probably the biggest different you will notice, other than having to use a different URL segments and ports, is that you will no longer run your runtime package, since doing so loads the EMS development server. Instead, you will compile your package, and then run whatever client you want to use to connect to your EMS endpoints.

## Restarting the EMS ISAPI DLL

Once you hit your EMS server DLL with IIS, IIS will have a hold on the DLL file, which in turn has loaded any custom packages you have registered. As a result, you will not be able to re-compile your runtime packages as long as IIS maintains this hold, which it will do until you restart the EMS server.

Similarly, as you learned earlier, any time you change the EMS configuration you must restart the EMS server. Doing so was easy with the console application. You simply closed it. With the EMS ISAPI DLLs, it is a little more involved.

There are two ways to restart the EMS ISAPI DLL servers. The first, and most desirable, is to recycle the application pool on which the DLLs are running. To do this, open the Internet Information Server (IIS) Manager from the Administrative Tools applet, expand the root node and select the Application Pools node. Right-click the application pool on which the ISAPI DLLs are running and select recycle:

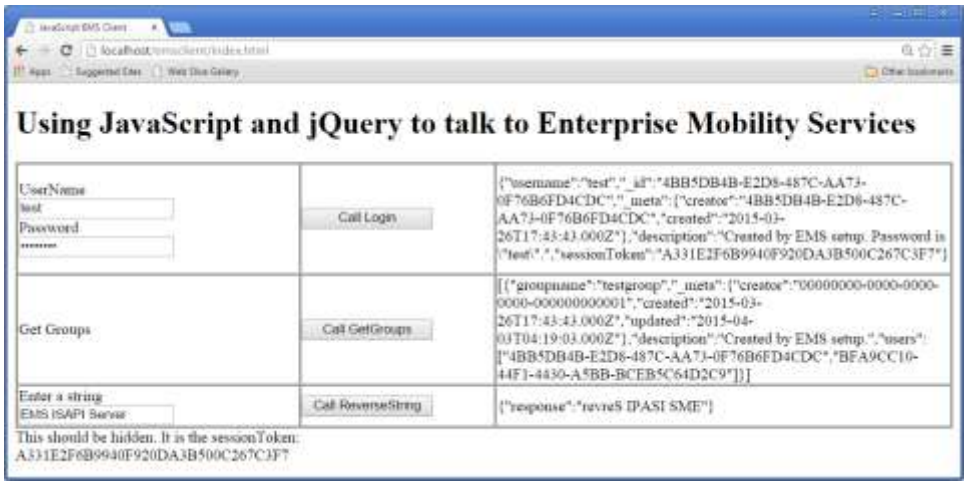



On rare occasions, recycling the application pool is not sufficient. In those cases, restart IIS. You can do this by selecting the root node in the Internet Information Server (IIS) Manager and then clicking Restart in the Actions pane.

## Calling EMS Endpoints from JavaScript/jQuery

Now that we have an ISAPI EMS server to use, we can invoke our EMS endpoints using JavaScript from a web page.

I have created a very simple web page that demonstrates how to login to the EMS server, and how to call some of its endpoints. My web page designer was out of the office the day that I created this demonstration, so I ask you to forgive the ham-handed design. This web page is shown in the following figure.




The source for this page can be found in a file named index.html, and which I placed into the following folder:

```
c:\inetpub\wwwroot\emsclient
```

This folder has a single subdirectory, into which I placed jQuery (version 1.12.2) and a custom JavaScript file that I created. This JavaScript file that creates a JavaScript object that exposes the functions that I use to call the EMS server.

The following is the content of index.html, which loads the two JavaScript files and wires my three EMS endpoint function calls to the click event of the three buttons shown on the html form:

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript EMS Client
</title>
<meta  http-equiv="Pragma" content="no-cache">
<meta http-equiv="refresh" content="1140">
<script type="text/javascript" src="./sharedscripts/jquery-
1.11.2.js"></script>
<script type="text/javascript"
src="./sharedscripts/EMSDemoEndpoints.js"></script>
<script>
$(function ()
{
  $("#loginBtn").click(ems.loginUser);
  $("#groupBtn").click(ems.getGroups);
  $("#reverse").click(ems.reverseString);
  ems.setBaseURL("http://localhost/emsserver/emsserver.dll/");
});
</script>
</head>
<body>
```

```
<h1 class="bod">Using JavaScript and jQuery to
                talk to Enterprise Mobility Services
</h1>
    <div class="contentdiv">
      <table border="1" cellspacing="1" cellpadding="1" width="100%"><tr>
            <td style="WIDTH: 378px">
              <p>UserName<br><input id="uname">
                <br>Password<br><input id="pword" type="password">
              </p></td>
            <td style="WIDTH: 240px"><input id="loginBtn" value="Call
Login"
                type="button" style="HEIGHT: 22px; WIDTH: 135px"
size="24"></td>
            <td>
              <div style="HEIGHT: 15px; POSITION: relative; DISPLAY:
inline;
                  WIDTH: 70px" id="loginResult"
ms_positioning="FlowLayout">
              </div>
            </td>
          </tr>
          <tr>
            <td style="WIDTH: 378px">Get Groups</td>
            <td style="WIDTH: 240px">
              <input id="groupBtn" value="Call GetGroups" type="button"
                    style="WIDTH: 135px"></td>
            <td>
              <div style="HEIGHT: 15px; DISPLAY: inline; WIDTH: 70px"
                   id="getGroupResult" ms_positioning="FlowLayout">
              </div></td>
          </tr>
          <tr>
            <td style="WIDTH: 378px">Enter a string<br><input
id="reversein"></td>
            <td style="WIDTH: 240px"><input id="reverse" value="Call
ReverseString"
                type="button" style="HEIGHT: 22px; WIDTH: 135px"
size="14"></td>
            <td>
              <div style="HEIGHT: 15px; DISPLAY: inline; WIDTH: 70px"
                   id="reverseout" ms_positioning="FlowLayout">
              </div>
            </td>
          </tr>
      </table>
</div>This should be hidden. It is the sessionToken:
<div id="sessionToken">
</body>
</html>
```

The following is EMSDemoEndpoints.js, the source file that demonstrates how to access the EMS server endpoints using JavaScript and jQuery. Each call is an asynchronous REST call using jQuery's ajax method. Anonymous methods are used to handle the result of each call, and a common error handler handles any errors.

Examples of both POST and GET calls are demonstrated here, which should give you clues as to how you can access your own EMS endpoints using JavaScript and jQuery.

A div, which should be hidden in practice, but which is visible on this web page for your benefit, is used to capture the session token after a successful login, and that token is returned to the EMS server on every call except for login. The prepHeader function is responsible for adding the session token to the request headers:

```
var com;

if (!com) com = {};
if (!com.jdsi) com.jdsi = {};
com.jdsi.emsendpoints = {};

//this is the global variable
ems = com.jdsi.emsendpoints;

(function () {
  //this base URL is using a relative path to avoid a cross domain issue
  var baseURL = "../../EMSServer.dll/";

    com.jdsi.emsendpoints.loginUser = loginUser;
    com.jdsi.emsendpoints.getGroups = getGroups;
    com.jdsi.emsendpoints.reverseString = reverseString;
    com.jdsi.emsendpoints.setBaseURL = setBaseURL;

    function loginUser(){
       var logdat = { username: $("#uname").val(),
                      password: $("#pword").val()};
      jQuery.ajax(
                   {type: "POST",
                   url: baseURL + "users/login",
                   data: JSON.stringify(logdat),
                   contentType: "application/json; charset=utf-8",
                   datatype: "json",
                   success:     function ( data, status, XHR) {
                                  var result;
                                  result =
jQuery.parseJSON(XHR.responseText);
                                  if (result.sessionToken) {
                                    token = result.sessionToken;
                                  }


$("#loginResult").text(XHR.responseText);
                                  if (! token == "") {
                                    $("#sessionToken").text(token);
                                  }
                                },
                   error: ferror});
    }

    function getGroups() {
      if ( isLoggedIn() )
```

```
        jQuery.ajax(
                    {type: "GET",
                    url: baseURL + "groups",
                    contentType: "application/json; charset=utf-8",
                    beforeSend: prepHeader,
                    datatype: "json",
                    success: function ( data, status, XHR) {

$("#getGroupResult").text(XHR.responseText);
                            },
                    error: ferror});
    }

    function reverseString() {
      if ( isLoggedIn() )
        jQuery.ajax(
                    {type: "GET",
                    url: baseURL + "Demo/ReverseString/" +
                                  $("#reversein").val(),
                    contentType: "application/json; charset=utf-8",
                    beforeSend: prepHeader,
                    datatype: "json",
                    success: function ( data, status, XHR) {
                              $("#reverseout").text(XHR.responseText);
                            },
                    error: ferror});
    }

    // Helper functions
    //
    //This function adds the sessionToken, if it has been captured
    function prepHeader( xhr ) {
      if ($.trim($("#sessionToken").text()) != "" ) {
        xhr.setRequestHeader("X-Embarcadero-Session-Token",
                            $("#sessionToken").text());
      }
    }
    //This function displays a dialog box if the user is not logged in
    function isLoggedIn() {
      if ($.trim($("#sessionToken").text()) != "" )
        return true
      else {
        alert("Please login");
        return false;
      }
    }

    function ferror(XHR, status, error) {
      $("echoout").text("error");
    }

    function setBaseURL(url) {
      baseURL = url;
    }
})();
```

# Enabling EMS Security

EMS provides for secure communications in two ways, the client/server interaction can be encrypted, and the individual clients can be challenged for a user name and password, ensuring that only authorized clients can communicate with the server. These features must be explicitly enabled, as EMS does not employ security by default.

## Encrypting Communications Using Secure Socket Layer

Encrypted communication is provided for by using SSL (Secure Socket Layer). If you are using the EMS ISAPI DLL, this is something that you configure from within IIS. Ideally, you will only permit HTTPS requests of your server. You do this by disabling HTTP for your EMS server connections.

If you want to encrypt your EMS development server communications, you edit the EMS configuration file, named emsserver.ini. This is the file that was created when you ran the EMS Setup Wizard. In that file you will find section named [Server.Connection.Dev].

To enable HTTPS, add the following name/value pair to the Server.Connection.Dev section:

```
HTTPS=1
```

You will also need to provide information about your SSL certificate. Normally you will obtain a certificate from a trusted certificate authority, such as Symantec, Thawte, or GoDaddy. Or, alternatively, you can generate a self-signed certificate. Doing so, however, will cause your browser to complain each time you hit your EMS server for the first time during a browser session.

Once you have your certificate, you can update the Server.Connection.Dev section of your configure to include CertFile and KeyFile values, and if you are using a certificate from a trusted certificate authority, a RootCertFile value. You can also provide a KeyFilePassword value if necessary.

## Authenticating Clients

Authentication is the process by which the EMS server requires client applications to provide information before permitting them to access EMS server endpoints. There are two

levels of authentication, depending on the needs of your service. These are application-level authentication and user-level authentication.

Application-level authentication requires that any client provide information before they can successfully request an endpoint, and EMS supports two types of application level authentication. These are AppSecret and MasterSecret, and they can be set in the Server.Keys section of the EMS configuration file.

By default, AppSecret is blank in the Server.Keys section. When AppSecret is set to a value, a client must provide the value of AppSecret in all requests for EMS resources. Clients pass the value of AppSecret in a request header, where the value part of the header is X-Embarcadero-App-Secret. When additional restrictions to resources and/or endpoints are defined in the Server.Authorization section of the EMS configuration file, both AppSecret, as well Server.Authorization restrictions, must be satisfied. Server.Authorization restrictions are discussed in the following section.

When MasterSecret is defined, and a client passes the value of MasterSecret in a request header, rights to the requested resource is granted unconditionally. Specifically, any restrictions defined in the Server.Authorization section of the EMS configuration file are ignored when a valid MasterSecret is provided. The value part of the MasterSecret header entry is X-Embarcadero-Master-Secret.

At the time of this writing, the generated EMS configuration file contains a value for the MasterSecret key. I strongly recommend that you either delete this value, or set it to something difficult to guess, in order to protect your EMS server's resources.

The Server.Keys section of the EMS configuration file also contains a key named ApplicationID. ApplicationID does not specifically apply to authentication. Instead, ApplicationID can be used in installations where there are multiple EMS servers, and can be used to avoid a mismatch between client and server. Clients pass the value of ApplicationID in each request header. The name part of the ApplicationID header is X-Embarcadero-Application-Id.

User-level authentication requires that a user formally log into the application, at which time they are provided with a session token. This session token, whose name is X-Embarcadero-Session-Token, should then be provide in the header of each subsequent request so that the EMS server can identify the user. This session token is the centerpiece of authorization.

# Authentication versus Authorization

Authentication is the process of identifying the user. To authenticate a user, a client must first call the Login or Signup endpoints of the User resource of the EMS administrative API. Login is a POST request that passes a JSON object in the HTTP message body. This object must have two JSON properties, "username" and "password." If the provided data matches that of an existing EMS user, a session token is returned to the client in the response.

Here is an example of a login request for the test user:

```
POST http://localhost:8080/users/login HTTP/1.0
Content-Type: application/json
Content-Length: 41
X-Embarcadero-Api-Version: 1
Connection: keep-alive
Host: localhost:8080
Accept: application/json, text/plain; q=0.9, text/html;q=0.8,
Accept-Charset: UTF-8, *;q=0.8
Accept-Encoding: identity
User-Agent: Embarcadero RESTClient/1.0

{"username":"test","password":"testpass"}
```

And here is the response to this successful call:

```
HTTP/1.1 201 Created
Connection: close
Content-Type: application/json; charset=ISO-8859-1
Content-Length: 271
Date: Sat, 28 Mar 2015 19:40:20 GMT
Location: http://localhost:8080/users/login/4BB5DB4B-E2D8-487C-AA73-
0F76B6FD4CDC

{"username":"test","_id":"4BB5DB4B-E2D8-487C-AA73-
0F76B6FD4CDC","_meta":{"creator":"4BB5DB4B-E2D8-487C-AA73-
0F76B6FD4CDC","created":"2015-03-
26T17:43:43.000Z"},"description":"Created by EMS setup.  Password is
\"test\".","sessionToken":"A331E2F6B9940F920DA3B500C267C3F7"}
```

Here is an example of a GET request for the users endpoint after this user has been logged in:

```
GET http://localhost:8080/users/ HTTP/1.1
X-Embarcadero-Api-Version: 1
X-Embarcadero-Session-Token: A331E2F6B9940F920DA3B500C267C3F7
Host: localhost:8080
Accept: application/json, text/plain; q=0.9, text/html;q=0.8,
Accept-Charset: UTF-8, *;q=0.8
```

```
Accept-Encoding: identity
User-Agent: Embarcadero RESTClient/1.0
```

Signup is a POST request that both creates a user and logs them in. Like a successful login, a Signup call will return the user's session token in the response. It is not necessarily the case that all applications will permit both login and signup. In fact, most systems will permit a user to login, but the creation of a new user is limited, by authorization, to a select group of users permitted to administer the application.

Authorization is the determination by the EMS server that a given user has the rights to invoke a particular endpoint. This can be done either at the user level or the group level. User level authorization is used to permit a specific user to access an endpoint or resource. Group level authorization permits users belonging to specific groups to access specific resources or endpoints.

You control authorization and authentication from the Server.Authorization section of the EMS configuration file. The entries that appear in this section are name/value pairs where the name identifies a resource or a resource endpoint, and the right side is a JSON object with a single property.

The property name is always one of the follow strings: "users", "groups", or "public". When the property is "users" or "group" the value is a JSON array of strings that identify the users or groups that are permitted to access the associated resource or endpoint. When the property is "public", the value is either true or false. When true, the resource or endpoint can be accessed by anybody. When false, the resource or endpoint cannot be accessed unless there is a further entry that explicitly grants rights to this resource to specific users or groups.

Let's look at a couple of examples. The following entry provides any client with access to the resource named Common. So long as there are no other entries in the Server.Authorization section related to Common or one of its endpoints, the absence of this entry also grants access to any client:

```
Common={"public": true}
```

While the following entry specifically prohibits access to the resource named Secret:

```
Secret={"public": false}
```

Endpoints are references using dot notation, where the resource name is followed by a dot, or period, and then by the endpoint name (or endpoint alias, if one has been

assigned using the EndpointName attribute in the endpoint declaration). For example, the following entry provides any client with access to the Login endpoint of the Users resource:

```
Users.Login={"public": true}
```

As mentioned, if neither a resource nor any of its endpoints appear in the Server.Authorization section, the resource is accessible to any client, so long as the AppSecret and ApplicationID requirements are met. This is the equivalent of using the following entry:

```
resourcename={"public": true}
```

If a resource, or at least one of its endpoints, is set to a JSON value whose property is "users" or "groups", it is the equivalent of setting that resource to the following:

```
resourcename={"public": false}
```

In other words, if you grant rights to a resource to either a collection of users (not ideal), or a collection of groups (much better), only those rights exist, and no rights are granted to any users or groups not listed.

Imagine that your application has, in addition to the three default, administrative resources, three more resources. Two of these, Customers and Accounts, are data related, and every authorized user should have access to the endpoints. The third, AccessControl, should be used only by a few authorized individuals.

Let's further assume that there are two groups defined for the EMS server: everyone and admin. Only those users assigned to the admin group should access the endpoints in the AccessControl resource.

Consider now the following Server.Authorization entries:

```
Users={"groups" : ["everyone"]}
Users.LoginUser={"public": true}
Users.AddUser={"groups", ["admin"]}
Users.DeleteUser={"groups", ["admin"]}
Groups={"groups" : ["everyone"]}
Groups.AddGroup={"groups", ["admin"]}
Groups.DeleteGroup={"groups", ["admin"]}
Groups.UpdateGroup={"groups", ["admin"]}
Customers={"groups" : ["everyone", "admin"]}
Accounts={"groups" : ["*"]}
AccessControl={"groups", ["admin"]}
```

The Version resource, which has a single endpoint, can be accessed by any client, since neither that resource nor its endpoint appears in the entries. Only members of the everyone group (which, by the way, admin members should also be a member of) can access the endpoints of the Users, Groups, Customers, and Accounts resources. I've included both the everyone and admin groups in the permissions for the Customers resource, simply to show how to include more than one group.

The second entry, in which Users.LoginUser is made public to everyone, permits any client to invoke login. If login is successful, that user will now be identified as a member of everyone, after which they can access all of the resources, except for those operations which should be restricted. Specifically, only members of the admin group can invoke the AddUser and DeleteUser endpoints of the Users resource, AddGroup, DeleteGroup, and UpdateGroup of the Groups resource, and all endpoints of the AccessControl resource.

A couple of final notes. If you intend to grant access to a resource or a specific endpoint to all users or all groups, you can simply use an asterisk. For example, in the above Server.Authorization entries, the entry for the Accounts resource is granted to all groups.

Also, it is possible to implement authorization programmatically from within your custom EMS endpoints. An example of this is demonstrated in the following code:

```
procedure TTestResource1.CheckAdministrator(const AContext:
TEndpointContext);
begin
  //Allow MasterSecret unconditionally
  if not (TEndpointContext.TAuthenticate.MasterSecret in
AContext.Authenticated) then
  begin
    if AContext.User = nil then
      EEMSHTTPError.RaiseUnauthorized('', 'User required');
    if not AContext.User.Groups.Contains('Administrators') then
      EEMSHTTPError.RaiseForbidden('', 'Administrator required');
  end;
end;

procedure TTestResource1.Get(const AContext: TEndpointContext;
  const ARequest: TEndpointRequest; const AResponse: TEndpointResponse);
begin
  CheckAdministrator(AContext);
  // implement response here
  // …
end;
```

# Using the BackendAuth Component

Earlier in this paper, you learned that you could use a variety of RAD Studio components to easily interact with your EMS server from RAD Studio applications. When your EMS server is requiring authentication, there is an additional component that you must use, and it is called BackendAuth. Like the other Backend components, a BackEndAuth component must reference a properly configured EMSProvider component in its Provider property.

You use the BackendAuth component to either login to or sign up with the EMS server. Both of these operations result in the corresponding user being logged in, and each of the requests is responded to with a message body that includes a JSON object from which a session token can be read. The BackendAuth component will manage this session token, and ensures that subsequent calls from the other Backend components include this session token in the request header of their REST invocations.

An example of how this works can be found in the EMSDemoClient project. Here is the code associated with the button labeled Login:

```
procedure TForm1.LoginLogoutBtnClick(Sender: TObject);
begin
  //You can also user BackEndUsers to log
  dm.BackendAuth1.Username := UsernameEdit.Text;
  dm.BackendAuth1.Password := PasswordEdit.Text;
  if not dm.BackendAuth1.LoggedIn then
    dm.BackendAuth1.Login
  else
    dm.BackendAuth1.Logout;
  UpdateLogin;
end;
```

The other backend components, such as BackendUsers, BackendGroups, BackendQuery, and BackendEndpoint, have an Auth property, to which you assign the BackendAuth instance that holds the user name and password, and, following a successful login, the session token. So long as all of the Backend components point to the same BackendAuth component, everything just works. Assuming, of course, that the user that is logged in, or the groups to which that user belongs, is authorized to access the endpoint that the Backend component attempt to request.

When your client applications are using Backend components, and employ application-level authentication, the ApplicationID, AppSecret, and MasterSecret values, when used, are assigned to properties of the EMSProvider component to which the BackendAuth component is wired. In addition, when MasterSecret is being used, the

Authentication property of the BackendAuth must be set to Root (the default value of TBackendAuth.Authentication is Default). Like with the session token, the BackendAuth will then ensure that the necessary application-level authentication headers are added to the EMS endpoint requests.

# Authentication without BackendAuth

Clearly you will not always use a BackendAuth component to manage logging in and accessing restricted resources (which require a valid session token). For example, any client application not written in RAD Studio simply does not have access to a BackendAuth.

Fortunately, this is not really a problem, and a solution to this has already been encountered earlier in this paper, in the code associated with the JavaScript/jQuery client. In short, any client can support authentication by performing the basic steps outlines in the EMSDemoEndpoints.js source code. These steps are:

1. Execute a valid HTTP POST request to the LoginUser or SignupUser EMS server administrative endpoint.
2. If successful, capture the session token from the JSON object contained in the response message body.
3. Submit that a session token in each subsequent HTTP request to the EMS server in a request header. If AppSecret, ApplicationID, and/or MasterSecret are defined for the EMS server, those headers must be submitted as well.

You already have one example of how this is done using JavaScript and jQuery. The EMSDemoClient application includes a second demonstration, in which RAD Studio's REST client library is employed. This example is nice because the REST client library is a general REST solution, and has no special connection with EMS services specifically.

The follow code has been adapted from the custom UpdateLogin method found in the EMSDemoClient project:

```
var
  s: string;
begin
  if dm.BackendAuth1.LoggedIn then
  begin
    if dm.BackendAuth1.LoggedInValue.TryGetAuthToken(s) then
    begin
      if dm.RESTClient1.Params.ParameterByName('X-Embarcadero-Session-
Token') = nil
      then
        dm.RESTClient1.AddParameter('X-Embarcadero-Session-Token', s,

TRESTRequestParameterKind.pkHTTPHEADER, [])
```

```
      else
        dm.RESTClient1.Params.
                        ParameterByName('X-Embarcadero-Session-
Token').Value := s;
    end;
//…
```

Subsequent calls to EMS server endpoints using RESTRequest components will succeed so long as their Client properties are set to the RESTClient to which the session token header has been added.

Sure, this code made use of the BackendAuth for the login, but a RESTRequest component could just as well have been used, in a manner similar to how the jQuery ajax method was used, to issue a POST request for the LoginUser endpoint, retrieving the session token from the response message body when the call succeeds.

# Using the EMS Console

The EMS console is a web-based application that makes calls into the EMS development console application or the EMS ISAPI DLL. And it is pretty easy to use, so long as you have a username and password for the web application. This user name and password is defined in the EMS configuration in the Console.Login section. Here is how this section looks in the default EMS installation:

```
[Console.Login]
UserName=consoleuser
Password=consolepass
```

To run the EMS development console, ensure that the EMS development console is running, and then enter the following URL into a browser:

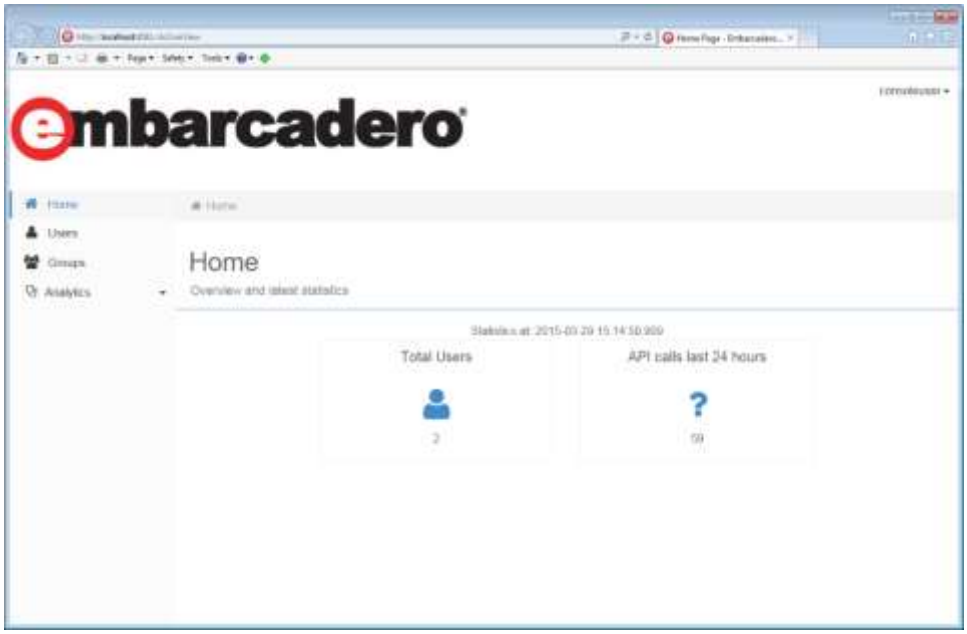
```
http://localhost:8081/
```

If you are using the EMS console ISAPI DLL, you use something like the following, which assumes that you've copied the emsconsole.dll file to the emsserver folder under c:\inetpub\wwwroot:

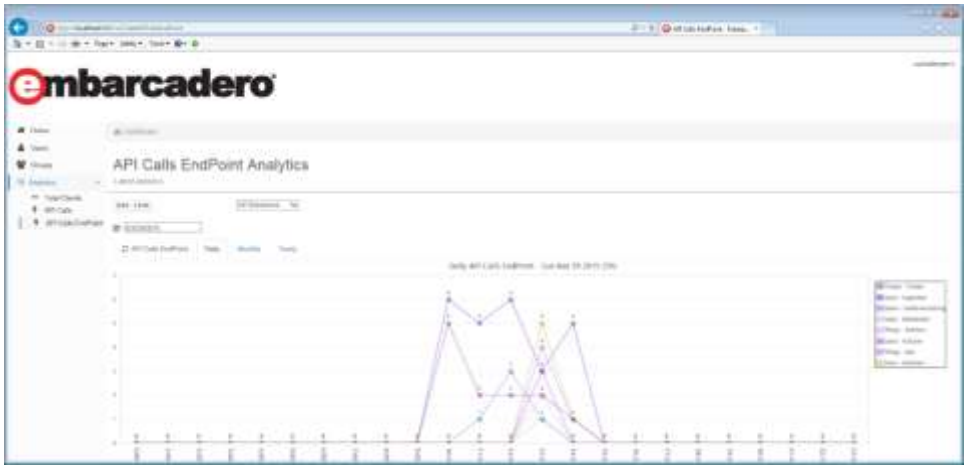
```
http://localhost/emsserver/emsconsole.dll
```

The EMS console login page is shown in the following figure:

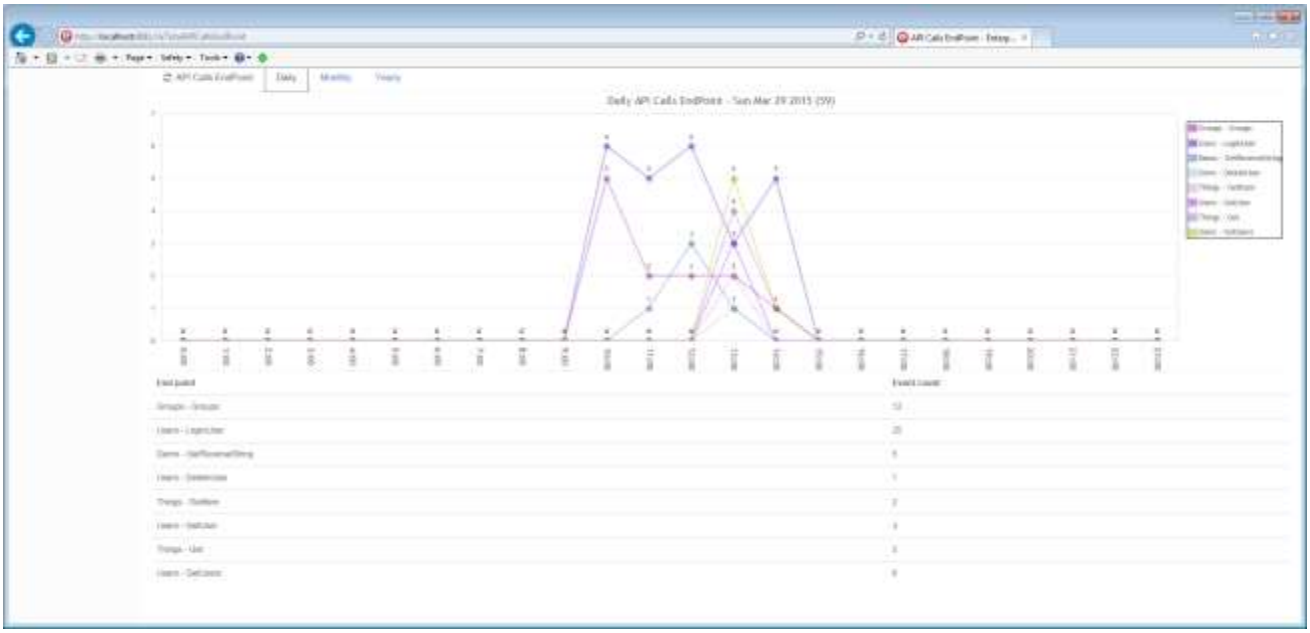

Click the Login link to display the login screen, and enter the EMS console user name and password, after which you should click the Login button:




The EMS console provides a number of links that you can click to see information about the usage of your EMS server. For example, you can use the options on the left-hand side of this web page to view information about users, groups, and endpoint statistics. In the following figure I have clicked on Analytics to expand that menu, after which I've clicked on the API Calls Endpoints link. The resulting figure shows the number of REST calls to my various endpoints over the past 24 hours, broken down by hour. If I click on the Monthly or Yearly tabs, I can see these breakdowns by day or month.




Scrolling down the page I can see the individual endpoints, and the number of calls each have received over the whole time period covered by the graph, as shown in the following figure:



Before continuing, I want to mention that the EMS console shown here is the one that shipped with RAD Studio XE7. Embarcadero has continued to improve the EMS console, exposing additional analytics and enhancing its features. In other words, your EMS console is likely to provide you with more information than that shown here.

# Debugging Custom EMS Resources

Debugging your custom EMS server endpoints is easier than you might expect, even if you are using the ISAPI DLL version. I will demonstrate debugging using both the EMS

development server as well as the ISAPI DLL version. The method that I am going to debug is the GetReverseString method that I referenced earlier in this paper in the section *Creating Additional Custom Endpoints*. If you did not add that code to your Demo resource, you will find a copy of that project in the code that accompanies this white paper.

Regardless of which version of the server you are running, you need two things. You need a host for your runtime package, and you need a client that can invoke the endpoint you want to debug. I've chosen the GetReverseString endpoint to debug, because it is associated with GET request, which means we can use a web browser to invoke the endpoint.

Also, before we start, place a breakpoint on the first line of the implementation of the GetReverseString method of the TDemoResource class.

Let's start by debugging with the EMS development server.

# Debugging Using the EMS Development Server

With the EMSDemoPkg project selected in the Project Manager, click Run (not Run without debugging) to launch the EMS development server. Even though your project is a DLL, and cannot be run directly, the debugger now appears to be active. This is because the debugger is attached to the host, which is the EMS development server.

Now, enter the following value into a browser:

```
http://localhost:8080/Demo/ReverseString/EMSServer
```

Once the browser makes the request of the EMS development server, that server will call the GetReverseString method of the TDemoResource1 class, and the breakpoint will cause execution to pause. You are now ready to step through this endpoint. In the following figure, I have clicked Step Over twice, and am now pausing my cursor above the LItem variable, which has caused the debugger to display the value of this variable:


That was easy, wasn't it?

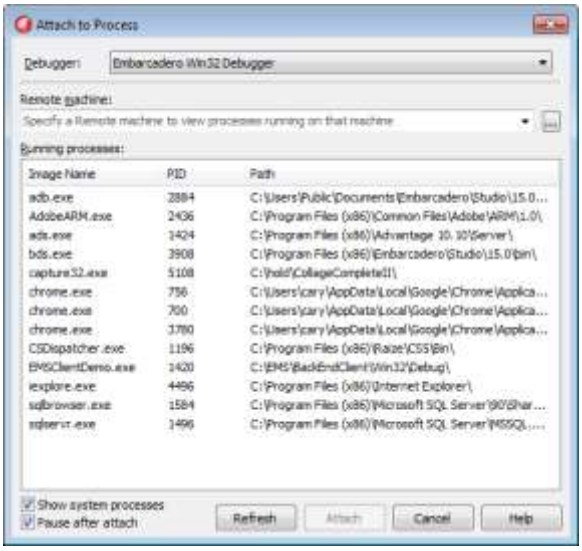
Let's now take a look at debugging this method when invoked by IIS.

## Debugging Using the EMS ISAPI DLL

When using the EMS ISAPI server, the host application is IIS, and this makes things only slightly more complicated. Here are two techniques. The first makes use of RAD Studio's attach to process capability to load IIS in the debugger.

Begin by making sure you still have the breakpoint that you placed in the preceding steps. Also, make sure that the host application isn't running. If it is, click the Reset button in RAD Studio, or select Run | Reset from RAD Studio's main menu, to close the host application.

Next, select Run | Attach to Process to display the Attach to Process dialog box shown here:

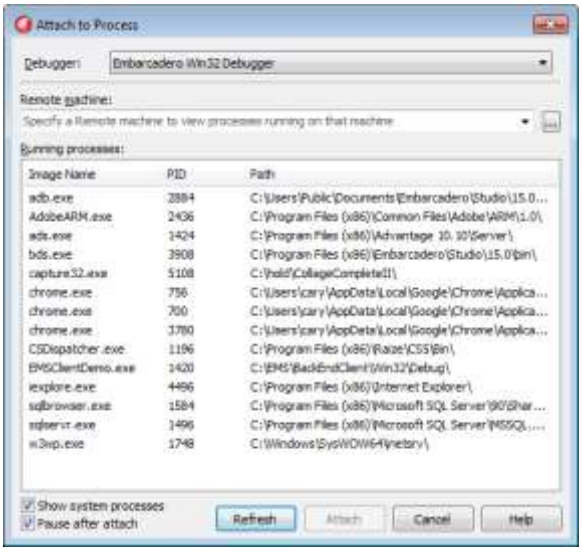



If IIS is running, we are ready to invoke the endpoint in which the breakpoint appears. If it is not, you have to load IIS first. The IIS process is called w3wp.exe, and it isn't loaded, as you can see in the preceding figure. Therefore, we need to cause it to load.

You cause IIS to load by hitting any resource. If you do not have any HTML files to load into IIS, just go ahead and hit the Version endpoint of the Demo resource. As you learned earlier, you can do this using the following URL:

```
http://localhost/emsserver/emsserver.dll/Version
```

Now, refresh the Attach to Process dialog box by clicking Refresh, and IIS will appear, as shown here:

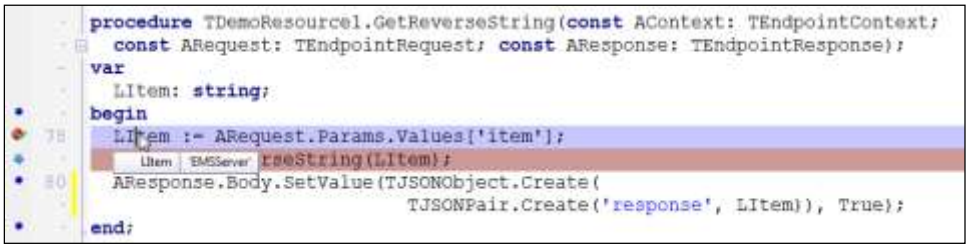



Good. Now, before continuing, uncheck the Pause after attach check box, if it is checked. Pausing after attaching is an annoying behavior, but it is the default.

Now, click on w3wp.exe and click Attach. IIS is now your host. You can now hit the GetReverseString endpoint by using the following URL in the browser:

```
http://localhost/emsserver/emsserver.dll/Demo/ReverseString/EMSServer
```

Once again, we are debugging GetReverseString. In the following figure I have stepped over once, and am showing the current value of the LItem variable:




During the technical review of this paper, Jim Tierney, Principal Engineer at Embarcadero Technologies, offered this easier alternative.

1. Stop IIS from the Internet Information Services Management Console, or enter the following statement from a command prompt that is running with administrative privileges:

   ```
   net stop w3svc
   ```

   You will have to do this each time you restart your operating system.

2. Select Run | Parameters from RAD Studio's main menu.

3. In the display dialog box, set Host to the fully-qualified path w3wp.exe. On my system it is C:\Windows\System32\inetsrv\w3wp.exe for 64bit or C:\Windows\SysWow64\inetsrv\w3wp.exe for 32bit.
4. Set Parameters to debug.
5. Click OK to save your settings.

Now, each time you run your EMS package, RAD Studio will launch IIS. No need to Restart IIS or recycle the application pool.

Great tip, Jim.

# EMS versus DataSnap REST

Enterprise Mobility Services and DataSnap are both middle-tier servers that you can deploy with RAD Studio, so they have a number of characteristics in common. They are, however, significantly different from the standpoint of how they work and how they are accessed by client applications.

EMS is a turnkey solution that provides for the secure management of user and groups, and authenticated access to REST endpoints. DataSnap is a general SDK (software development kit) that enables you to build middle-tier applications. It has a rudimentary framework for authentication and authorization, but you must implement the details manually.

EMS is a truly stateless middle-tier server. DataSnap, on the other hand, provides a lifecycle option, invocation, that simulates a stateless connection, but it is stateful by default. For example, when authorization is enabled, an EMS client can connect to an EMS server successfully, without re-authorizing, even if the server is behind a load balancer, and each request may be served by a different instance of the EMS server. Similarly, an EMS client that has been authenticated has no issues connecting to an EMS server, ever after the server was restarted since authentication took place.

By comparison, when a DataSnap server requires authentication, and lifecycle is set to invocation, authentication will occur on every resource request, which requires two round trips. If another DataSnap lifecycle option is used, the client will have to be re-authenticated after a server-side restart.

EMS REST interfaces conform to industry standards for REST APIs, while DataSnap is somewhat more idiosyncratic. With DataSnap, these idiosyncrasies can be accommodated, if you want, but that requires adding additional, custom code.

On the other hand, DataSnap supports a non-REST interface that permits it to easily expose DataSetProviders on the server for consumption by ClientDataSets on client application. These clients, however, are strictly RAD Studio clients, in that any client of this interface must be an executable compiled by RAD Studio. EMS clients can be created by any framework that supports HTTP request and response.

EMS is a REST service, and your clients communicate with it using HTTP or HTTPS. DataSnap supports both HTTP and HTTPS traffic, but supports TCP/IP (Transmission Control Protocol/Internet Protocol) as well. TCP/IP is generally faster than HTTP, making it more efficient when large amounts of data need to be transferred.

EMS can be implemented using an Enterprise edition or higher of RAD Studio, Delphi or C++Builder, or a Professional edition to which the FireDAC Client/Server Pack has been added. DataSnap requires an Enterprise version of RAD Studio or higher.

The final difference is that a license must be purchased to deploy an application using EMS, while DataSnap can be deployed royalty-free. It's worth noting that an EMS license also comes with a license for InterBase server, as well as InterBase ToGo licenses for EMS clients. If you want to use InterBase with DataSnap, a separate InterBase license must be purchased.

# Upcoming Features

The features covered so far are all part of the first version of EMS delivered with RAD Studio XE7. At the time of this writing, the next version of RAD Studio has not been released, but there is some information about features that are being added to EMS in the next release, and that have partially been demonstrated in a recent Skills Sprints webinar. This section of this white paper introduces some of these upcoming features of EMS.

The most relevant addition to the second version of EMS will be the support for Push Notifications. This is a mobile technology that allows a server to send a notification to an application installed in a phone or tablet client using either iOS or Android, using specific services provided by the respective operating system vendors, which are Apple and Google, respectively. Push notifications show up like app notifications, and can be sent to the device and seen by the user even if the application is not currently running. The Push Notification carries extra content (in the form of a JSON data structure) that can be used to instruct the app what to do when the application is received, either directly as the application is running or upon application startup if the application is not running and the user clicks on the notification itself to open the app.

Push Notifications support is exposed by EMS as a new custom resource: calling that EndPoint (wrapped in a specific component) will allow a client application (and also code running within an EMS module) to trigger a notification for a specific user's device, all users who subscribed to a specific resource, or broadcast to all users registered in the EMS server. There is also a ready-to-use component for receiving notifications in a mobile FireMonkey application.

In the second version of EMS, the EMS Console has been expanded to capture more user and group analytics and export the analytics data in a CSV format, which can later be imported by many external tools for custom processing. Also, there is a new client version of the console for managing groups, users, and Push Notifications, that can be used for easily importing and exporting users and for testing notifications without having to write specific code to trigger them.

Another new feature important for enterprise customers is the ability to plug a custom authentication module into an EMS server so that the users's passwords can be validated against an external service rather than stored directly into the EMS database. This allows, for example, you to use Microsoft's ActiveDirectory authentication for EMS users, a scenario covered by a specific example that ships with the new version of EMS.

Other extensions include database connection pooling support and new features in the Administrative API. There is also a new client component to further simplify writing EMS client code. More information will become available as the new version of EMS ships.

# Summary

Enterprise Mobility Services (EMS) is a turnkey middle-ware solution that supports secure, encrypted communication using an industry-standard REST interface. With built-in support for users and groups, EMS is a perfect platform for exposing your custom REST endpoints to a wide range of authorized clients. Developers will especially appreciate the ease with which corporate data and features can be securely accessed from mobile clients, which normally lack the ability to easily interface with enterprise data.

# Sample Projects

The sample projects discussed in this paper should be found in the same compressed file in which you found this white paper. In case you cannot find these projects, I have also made them available from the following URL:

```
http://www.JensenDataSystems.com/EMSWhitePaper
```

# Acknowledgements

I want to express my thanks for the generous help of the many people who were involved in the production of this white paper and the associated Webinar. In particular, I want to thank Jim Tierney, Principle Engineer at Embarcadero Technologies, for his technical support, several code samples, and incisive input on a draft of this paper. Similar kudos go to Marco Cantú, RAD Studio Product Manager, for his technical support and review of the paper. I also want to thank John Thomas (JT), for asking me to write this paper, his marketing insight, and for ensuring that I got the technical support that I needed. I also want to thank my wife and business partner, Loy Anderson, who also reviewed the final draft of this paper. Finally, I want to thank the many other people at Embarcadero Technologies who helped this project at various stages, including Tim Del Chiaro, Brian Alexakis, David Intersimone (David I), and Jim McKeeth.

# About the Author

Cary Jensen is Chief Technology Officer of Jensen Data Systems, Inc., a Texas, USA-based company that is an Embarcadero Technology Partner and that provides training, development, and consulting services. Cary is an Embarcadero MVP and an award-winning, best-selling author of over two dozen books on software development, including his latest book, Delphi in Depth: ClientDataSets 2nd Edition. He has written hundreds of magazine articles, and is a popular speaker at conferences, workshops, and training seminars around the world, and is widely regarded for his practical solutions to complex problems. Cary wrote, and was the principal speaker for the original Delphi World Tour in 1995, and in 2001 he founded Delphi Developer Days, a popular training seminar that visits cities in North America and Europe each spring. Cary has a Ph.D. in Engineering Psychology from Rice University, specializing in human-computer interaction. You can learn more about Cary at www.JensenDataSystems, and follow him on twitter (@caryjensen).