

Building and Consuming REST Services with FoxPro

REST APIs, or Web Services that use plain HTTP requests and JSON have become the replacement for more complex SOAP based service architectures of the past. Most modern APIs available on the Web — from Credit Card Processors, to eCommerce back ends, to mail services, Cloud Provider APIs and Social Media data access — all use REST services or variants thereof to make remote data available for remote interaction.

REST services tend to be much simpler to build and consume than SOAP, because they don't require any custom tooling as SOAP/WSDL services did. They use the HTTP protocol for sending requests over the Web, and typically use JSON's as their serialization format. JSON's simple type structure is inherently easier to create and parse into object structures from a language like FoxPro and REST's clear separation between the message (JSON) and the protocol layers (HTTP Headers/Protocol) reduces the amount of infrastructure that is required in order to use the technology.

Because of its simplicity REST can also be directly consumed by Web applications rather than going through a server proxy. JSON is a JavaScript native format (essentially an *object literal*) and so any JavaScript applications can easily consume REST services directly.

This makes REST useful for double duty both as a remote data service API and a backend for internal SPA type Web applications. Often these two tasks can overlap, with applications exposing both the Web application for interactive Web and App use, and a service for remote data API access. Many big services like Twitter, Facebook and Cloud Providers like Azure use APIs to drive their front ends while also exposing those same APIs for remote access.

One of the big reasons of REST's popularity and success in recent years is its simplicity: All you need to consume a REST Service is an HTTP Client and a JSON parser. On the server too no special tools are required beyond a Web Server and the ability to capture HTTP requests and write HTTP responses which means that its easy to create REST service endpoints manually, and there are lots of support frameworks to choose from to provide automated REST service integrations.

Client and Server

For this article there are two scenarios that you're going to deal with when it comes to REST Services:

- Consuming REST Services
- Creating Server APIs using REST Services

I'll talk about both of these scenarios in the context of Visual FoxPro. We'll start with retrieving some data from an HTTP service and consuming it in FoxPro, and then jump to the other end and create a REST JSON service on the server side using [Web Connection](#).

But before the practical bits, let's talk about what REST is and what makes it unique and how it differs from what came before.

So what is REST?

Unlike older protocols like SOAP, . REST is not a specific standard or even a specification and it doesn't have a formal definition. There's no Web site that you can go to to look up how to specifically architect your HTTP Service. Rather it's a set of common '*recommendations*' or a '*style*' of building HTTP based Web Services based on the semantics of the HTTP protocol.

Officially REST stands for **Representational State Transfer** which is a fairly cryptic term to describe what amounts to Web based APIs. The idea behind the term is that you have fixed URLs from which you can transfer state - or data - back and forth between a client and server.

Since there isn't a fixed standard you can look at, here's Wikipedia's broad definition:

Representational state transfer (REST) is a software **architectural style** that was created to guide the design and development of the architecture for the World Wide Web. REST defines a **set of constraints** for how the architecture of an Internet-scale distributed hypermedia system, such as the Web, should behave. The REST architectural style emphasizes the scalability of interactions between components, uniform interfaces, independent deployment of components, and the creation of a layered architecture to facilitate caching components to reduce user-perceived latency, enforce security, and encapsulate legacy systems.

This is pretty vague and open to interpretation with words like **architectural style** and **general set of constraints**. There's nothing specific about this 'recommendation', other than it uses the HTTP protocol to access and send data.

REST is all about HTTP

REST is all about taking maximum advantage of the HTTP Web protocol.

HTTP is the protocol used to communicate on the Web. HTTP traditionally has been the protocol of Web Browsers, but more recently the use of Web APIs increasingly sees HTTP use by applications using HTTP client software either built into frameworks or tools.

HTTP is very prominently used in today's modern applications even outside of the context of traditional Web applications: You see APIs used heavily these days in native Mobile apps as well as many desktop applications.

The HTTP protocol is used to send and retrieve data in a simple, one-way transactional manner: A request is made with headers and content, and a response is returned also with headers and content.

Requests only go one way from the client to the server. While the server can return data from in response to a request, it cannot independently call back to the client outside of an incoming request context. There are other ways to do this namely using Web Sockets that are built on top of HTTP, but that's a separate protocol and not applicable to REST.

Finally it's important to remember that HTTP is inherently stateless - each request has to provide its own context to the server as each request opens and closes a connection to the server. There's no explicit persistent state across requests unless some mechanism like HTTP Cookies or custom headers are used between requests. It's unusual though to use these mechanisms for APIs - API clients tend to keep state in the context of the application and then send it as part of the request or the request headers which most commonly includes authentication in the form of auth tokens.

Here's what the HTTP Request and Response are made up of:

Request

- HTTP Host and Path (Url)
- HTTP Verb (GET , POST , PUT , DELETE , OPTIONS)

- HTTP Request Headers
- Request Body (for `POST` and `PUT`)
- Request Body is usually JSON
but can be any other raw data (xml, binary, form data)

Response

- HTTP Response Headers
- Response Body

Here's what a real HTTP request looks like. This first example is a simple `GET` request that only retrieves JSON data from a server:

Request Headers

Copy
 Run

GET https://albumviewer.west-wind.com/api/artist/33 HTTP/1.1

Accept-Encoding: gzip,deflate
Accept: application/json
User-Agent: West Wind HTTP .NET Client

Response Headers

3,442 bytes
Time: 52ms
First byte: 52ms

HTTP/1.1 200 OK

Server: Microsoft-IIS/10.0
X-Powered-By: ASP.NET
Date: Fri, 24 Sep 2021 21:31:44 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 3442

Response Content

```
{
  "Artist": {
    "Id": 33,
    "ArtistName": "Anti-Trust",
    "Description": "Anti-Trust is a side project by ex-Attitude Adjustment members Chris Kontos, Rick Strahl and Andy Andersen. This collaboration produced a handful of songs that were garage recorded in Oakland, CA and Maui, HI in 2001 and 2002 by Rick and Chris. Several additional songs were recorded in late 2020 and early 2021 which resulted in a the songs being officially put out and released online and in album form.\n\nAnti-Trust's music features diverse influences from old school hardcore punk rock, metal cross over and NWOFBHM, all driven by heavy guitar rhythms and catchy choruses with a unique and edgy sound.",
    "ImageUrl": "https://anti-trust.rocks/images/Photo6.jpg",
    "AmazonUrl": "https://anti-trust.rocks"
  },
  "Albums": [
    {
      "Id": 37,
      "Title": "Guilty",
      "Description": "Old school hardcore punk with metal roots, kicked out in good
```

`GET` describes the HTTP verb used against the URL which retrieves an Artist instance. `GET` is a retrieval only request and the server returns an HTTP response, which is a nested JSON object.

Any REST request lives at a fixed URL which is unique, and is accessed via an HTTP Verb - `GET` in this case. The combination of URL plus HTTP Verb make for a unique resource that can be easily linked to or bookmarked in browsers.

Commonly used verbs are `GET`, `POST`, `PUT`, `DELETE`, `OPTIONS` which describe an 'action' on the resource you are accessing. Multiple verbs are often overloaded on a single URL that have different behavior depending on using a `GET` to retrieve and Artist for example, or `POST` / `PUT` to add or update and `DELETE` to delete.

The response in this example returns the requested Artist as a JSON (`application/json`) response. The response consists of HTTP headers that describe protocol and content information from the server and can also be used to send non-data related meta-data from the application to the client.

The second example, uses a `POST` operation to add/update an Artist which looks similar but adds Request Content to send to the server:

The screenshot displays a REST client interface with three main sections: Request Headers, Request Content, and Response Headers.

Request Headers: The method is `POST` and the URL is `https://albumviewer.west-wind.com/api/artist`. The headers are:

- `Accept: application/json`
- `Accept-Encoding: gzip, deflate`
- `Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV`
- `User-Agent: West Wind HTTP .NET Client`
- `Content-Type: application/json`

Request Content: The content is formatted JSON:

```
{
  "Id": 33,
  "ArtistName": "Anti-Trust",
  "Description": "!!! Anti-Trust is a side project by ex-Attitude Adjustment members Chris Kontos, Rick Strahl and Andy Andersen. This collaboration produced a handful of songs that were garage recorded in Oakland, CA and Maui, HI in 2001 and 2002 by Rick and Chris. Several additional songs were recorded in late 2020 and early 2021 which resulted in a the songs being officially put out and released online and in album form.\n\nAnti-Trust's music features diverse influences from old school hardcore punk rock, metal cross over and NWOFBHM, all driven by heavy guitar rhythms and catchy choruses with a unique and edgy sound.",
  "ImageUrl": "https://anti-trust.ocks/images/Photo6.jpg",
  "AmazonUrl": "https://anti-trust.ocks"
}
```

Response Headers: The status is `HTTP/1.1 200 OK`. The headers are:

- `Server: Microsoft-IIS/10.0`
- `X-Powered-By: ASP.NET`
- `Date: Fri, 24 Sep 2021 20:40:24 GMT`
- `Content-Type: application/json; charset=utf-8`

Additional response details shown are: 3,450 bytes, Time: 202ms, First byte: 202ms.

```
Content-Length: 3450

Response Content

{
  "Artist": {
    "Id": 33,
    "ArtistName": "Anti-Trust",
    "Description": "!!! Anti-Trust is a side project by ex-Attitude Adjustment members Chris Kontos, Rick Strahl and Andy Andersen. This collaboration produced a handful of songs that were garage recorded in Oakland, CA and Maui, HI in 2001 and 2002 by Rick and Chris. Several additional songs were recorded in late 2020 and early 2021 which resulted in a the songs being officially put out and released online and in album form.\n\nAnti-Trust's music features diverse influences from old school hardcore punk rock, metal cross over and NWOFBHM, all driven by heavy guitar rhythms and catchy choruses with a unique and edgy sound.",
    "ImageUrl": "https://anti-trust.rocks/images/Photo6.jpg",
    "AmazonUrl": "https://anti-trust.rocks"
  },
  "Albums": [
    {
```

This particular request is an update operation that updates an Artist in a music store application.

The POST operation is different in that it uses the `POST` verb, and provides a content body that contains the JSON request data. The data sent can be raw data like a JSON or XML document, but can also be Urlencoded form data, a multipart form upload, raw binary data like PDF or Zip file... it can be anything. Whenever you send data to the server you have to specify a `Content-Type` so that the server knows how to handle the incoming data. Here the data is JSON so `Content-Type: application/json`.

The HTTP Headers provide protocol instructions and information such as the calling browser, what type of content is requested or optionally sent, and so on. Additionally you can also handle security via the `Authorization` header. This example uses a **Bearer Token** that was previously retrieved via a `Authentication` API call. Headers basically provide meta data: Data that describes the request, or additional data that is separate from the data in the content of a request.

`POST` and `PUT` requests like also have a request body, which is raw data sent to the server. The data sent is serialized JSON of an Artist object to update the server with.

HTTP Advantages

HTTP is a great mechanism for applications because it provides many features 'out of the box' that don't have to be implemented for each tool or application.

Here are a few things that REST can take advantage of with HTTP:

- API Routing via URL
- Unique Resource Access via URL
- API Operations via HTTP Verbs
- Data Encryption via HTTPS (TLS)
- Caching via built-in HTTP Resource Caching

- Authorization via HTTP Authorization (+server auth support)
- Meta Data via HTTP Headers

HTTP Routing plus HTTP Verbs

HTTP has an innate built-in unique routing mechanism based on URLs. Any URL by its nature is a unique identifier, so each endpoint you create via an HTTP API is always unique in combination with an HTTP Verb.

A url like:

```
https://albumviewer.west-wind.com/api/artists GET
```

is 100% unique.

So are these even though they point at the same URL:

```
https://albumviewer.west-wind.com/api/artists POST  
https://albumviewer.west-wind.com/api/artists DELETE
```

Same URL different action, but different actions that are interpreted separately in the server application.

There are quite a few Verbs available and each has a 'suggested' meaning.

- **GET**: Retrieve data
- **POST**: Add data
- **PUT**: Update data
- **DELETE**: Delete data
- **OPTIONS**: Return headers only
- **PATCH**: Partial Update

Of these **POST** and **PUT** are the only ones that support a content body to send data to the server. All others are either data retrieval or operation commands.

These verbs are **suggestions**. Requests are not going to fail if you update data via a **POST** operation instead of using the suggested **PUT** unless the server applications explicitly checks and rejects requests based on a verb. However, it's a good idea to follow these **suggestions** as best as possible for consistency, and easy understanding of your API and when necessary make them flexible so they just work. It'll make your API easier to use.

Encrypted Content via HTTPS

HTTP has built in support for **https://** which uses certificate based security keys for encrypting content between client and server. This encryption ensures that content on the wire is encrypted and can't be spied upon without access to the keys of the certificates on both sides of the connection. This avoids man in the middle attacks. To use **https://** encryption a server secure certificate is required but these days you can set up free LetsEncrypt Certificates on most Web servers in minutes. For Windows Server and IIS look at **Win-Acme** to set up Lets Encrypt certificates on IIS for free.

The nice thing with **https://** is that it's part of the server infrastructure. As long as the server has a certificate,

both client and server can use the `https://` protocol to securely access requests.

Resource Caching

Each URL + Verb on an API endpoint is unique in the eyes of the browser and if you access the same resource using a read (ie. `GET`) operation, requests are cached on subsequent access by default. By default requests are expected to be idempotent which means that sending a request in the same way twice should always produce the same result. HTTP provides this functionality by default, but it can be overridden with specific HTTP headers that force the client to refresh data. This makes sense in some cases where data changes frequently.

Authorization and Authentication

HTTP doesn't have direct support for authentication besides the `Authorization` header that is commonly used by server frameworks to handle Authorization and Authentication. Most server frameworks today have some basic mechanisms for handling security built-in. Most Web Servers have support for Basic Authentication out of the box, IIS additionally has support for Windows Auth, and if you use an application framework like ASP.NET MVC or ASP.NET Core they also have built-in support for handling Cookie and Bearer token authentication as well as various federated frameworks.

Meta Data in HTTP Headers

Unlike SOAP, REST clearly separates the content from meta data that describes the request or response. So the content sent and returned tends to be truly application specific while anything that involves the request processing or tracking generally is handled in the headers of the request.

Every request has a handful of required headers that are always sent by the client and are always returned by the server. These describe the basics of the request or response and include things like the content type, content-length, the accepted types of content, browser and so on.

But beyond the auto-generated headers, you can also add custom headers of your own to both the client request and the server response. You should use headers to return data that is important to the application, but not directly part of the data. This could be cached state (similar to cookies) that you carry from request to request, or identifying information.

Calling REST APIs from FoxPro

Ok - enough theory let's kick the tires and use some RESTful APIs. Let's start with what's required to call a REST service from Visual FoxPro:

- **HTTP Client**
 - `wwHttp` (West Wind Tools)
 - `WinHttp` (GitHub Example)
 - `MSXML.ServerXMLHttp`
 - `curl`
- **JSON Parser**
 - `wwJsonSerializer` (West Wind Tools)
 - `nfJson`

Http Client

There are a lot of options for HTTP access. I'm obviously biased towards the `wwHttp` library as that's what I usually use and as it provides full featured HTTP support for many different scenarios. That's what I'll use for the examples here and the support libraries are provided with the samples.

A simple WinHttp Client

If you'd rather use a native tool without extra dependencies you can use **WinHttp** which is built into Windows. It has both Win32 and COM APIs. Using the COM API here's a very simplistic, generic HTTP client you can use instead of `wwHttp` :


```

*****
* WinHttp
*****

FUNCTION WinHttp(lcUrl, lcVerb, lcPostData, lcContentType)
LOCAL lcResult, loHttp

IF EMPTY(lcUrl)
    RETURN null
ENDIF
IF EMPTY(lcVerb)
    lcVerb = "GET"
    IF !EMPTY(lcPostData)
        lcVerb = "POST"
    ENDIF
ENDIF

*** Example of using simplistic WinHttp client to retrieve HTTP content
LOCAL loHttp as WinHttp.WinHttpRequest.5.1, lcResult
loHTTP = CREATEOBJECT("WinHttp.WinHttpRequest.5.1")

loHTTP.Open(lcVerb, lcUrl,.F.)

IF !EMPTY(lcContentType) AND lcVerb = "POST" OR lcVerb = "PUT"
    loHttp.setRequestHeader("Content-Type",lcContentType)
ENDIF

*** If using POST you can post content as a parameter
IF !EMPTY(lcPostData)
    loHTTP.Send(lcPostData)
ELSE
    loHttp.Send()
ENDIF

lcResult = loHttp.ResponseText

loHttp = NULL

RETURN lcResult

```

You can use it with very simple code like this:

```
SET PROCEDURE TO WinHttp ADDITIVE
lcResult = WinHttp("https://albumviewer.west-wind.com/api/artist/1")
? PADR(lcResult,1000)

TEXT TO lcJson NOSHOW
{
  "username": "test",
  "password": "test"
}
ENDTEXT
lcResult = WinHttp("https://albumviewer.west-wind.com/api/authenticate","POST",;
                  lcJson,"application/json")
? lcResult
```

This is a pretty basic implementation. It needs additional error handling, dealing with binary data, progress handling and a few other things, but for starters it's a workable solution.

wwHttp - A little Extra

The `wwHttp` provides a lot more functionality out of the box. It supports a number of convenience helpers to make it easy to parse both content and headers, encode and decode content, progress events, handle Gzip/Deflate compression, binary content, status updates and more. A compiled version of `wwHttp` is provided with the samples.

Using the same service as above using `wwHttp` looks something like this:

```
DO wwHttp && load libraries
```

```
loHttp = CREATEOBJECT("wwHttp")  
lcResult = loHttp.Get("https://albumviewer.west-wind.com/api/artist/1")  
? PADR(lcResult,1200)
```

```
TEXT TO lcJson NOSHOW
```

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

```
ENDTEXT
```

```
loHttp = CREATEOBJECT("wwHttp")  
loHttp.cContentType = "application/json"  
lcResult = loHttp.Post("https://albumviewer.west-wind.com/api/authenticate",lcJson)
```

```
IF loHttp.nError # 0  
  ? loHttp.cErrorMsg  
  RETURN  
ENDIF  
IF loHttp.cResultcode # "200"  
  ? "Invalid HTTP response code: " + loHttp.cResultCode  
ENDIF
```

```
? lcResult
```

JSON Serialization and Parsing

Next you need a JSON serializer that can turn your FoxPro objects into JSON, and turn JSON back into FoxPro objects, values or collections. I'm going to use `wwJsonSerializer` here since that's all I use, but there are other open source libraries available as well. The logic should be similar.

Objects, Values and Collections

Using `wwJsonSerializer` to turn a FoxPro object into a JSON looks something like this:

DO wwJsonSerializer && Load libs

*** Create a complex object

```
loCust = CREATEOBJECT("Empty")
ADDPROPERTY(loCust,"Name","Rick")
ADDPROPERTY(loCust,"Entered",DATETIME())
```

*** Create a nested object

```
ADDPROPERTY(loCust,"Address", CREATEOBJECT("Empty"))
ADDPROPERTY(loCust.Address,"Street","17 Aluui Place")
ADDPROPERTY(loCust.Address,"City","Paia")
ADDPROPERTY(loCust,"Number",32)
```

```
loSer = CREATEOBJECT("wwJsonSerializer")
```

*** Serialize into JSON

```
lcJson = loSer.Serialize(loCust)
```

? lcJson

*** read back from JSON into an object

```
loCust2 = loSer.DeserializeJson(lcJson)
```

? loCust2.Name

? loCust2.Entered

? loCust2.Address.Street

? loCust2.Number

This creates JSON like this:

```
{
  "address": {
    "city": "Paia",
    "street": "17 Aluui Place"
  },
  "entered": "2021-09-25T01:07:05Z",
  "name": "Rick",
  "number": 32
}
```

Simple Values

JSON has literal values for simple types and you can serialize and deserialize these simple values.

```
? loSer.Serialize("One" + CHR(10) + "Two" +;   && "One\nTwo\nThree"
      Chr(10) + "Three")
? loSer.Serialize(1.22)           && 1.22
? loSer.Serialize(.T.)           && true
? loSer.Serialize(DateTime())    && "2020-10-01T01:22:15Z"

*** Binary Values as base64
? loSer.Serialize( CAST("Hello World" as Blob)) && "SGVsbG8gV29ybGQ="
```

Collections and Arrays

Single dimension arrays and collections are supported for serialization. This is common for serializing object arrays, or just representing db records as objects for example.

```
loSer = CREATEOBJECT("wwJsonSerializer")

loCol = CREATEOBJECT("Collection")

loCust = CREATEOBJECT("Empty")
ADDPROPERTY(loCust,"Name","Rick")
ADDPROPERTY(loCust,"Company","West Wind Technologies")
ADDPROPERTY(loCust,"Entered",DATETIME())

loCol.Add(loCust)

loCust = CREATEOBJECT("Empty")
ADDPROPERTY(loCust,"Name","Kevin")
ADDPROPERTY(loCust,"Company","OakLeaf")
ADDPROPERTY(loCust,"Entered",DATETIME())
loCol.Add(loCust)

? loSer.Serialize(loCol, .T.)
```

The result is a top level JSON array of objects:

```
[
  {
    "company": "West Wind Technologies",
    "entered": "2021-09-25T05:12:55Z",
    "name": "Rick"
  },
  {
    "company": "OakLeaf",
    "entered": "2021-09-25T05:12:55Z",
    "name": "Kevin"
  }
]
```

Cursors

You can also serialize Cursors which are serialized as JSON object arrays similar to the last example. wwJsonSerializer uses a special string syntax to pull in a cursor or table by alias name using `cursor:TCustomers` syntax.

This first example is a top level cursor serialization:

```
loSer = CREATEOBJECT("wwJsonSerializer")

SELECT * FROM CUSTOMERS ORDER BY LastName INTO CURSOR TQuery

*** Serialize a top level cursor to a JSON Collection
lcJson = loSer.Serialize("cursor:TQuery")
? PADR(lcJson,1000)
```

This produces a top level array:

```
[
  {
    "id": "_4FG12Y7TK",
    "firstname": "Pat",
    "lastname": "@ Accounting",
    "company": "Windsurf Warehouse SF",
    "careof": "Pat @ Accounting",
    "address": "405 South Airport Blvd. \r\nSouth San Francisco, CA 94080",
    "entered": "2014-11-02T10:46:40Z",
    "state": "OR"
  },
  {
    "id": "_4FG12Y7U7",
    "firstname": "Steven",
    "lastname": "Black",
    "company": "SBC",
    "careof": "Steven Black",
    "address": "12 East Street\r\nKingston, ON\r\nK7K 6T3 Canada\r\n",
    "entered": "2014-06-02T09:46:40Z",
    "state": ""
  },
  ...
]
```

The second example, creates a cursor as a nested object collection in a top level object as property and setting the to `cursor:TCustomers` :

```
*** Cursor as a Property of a complex object
loCust = CREATEOBJECT("Empty")
ADDPROPERTY(loCust,"Name","Rick")
ADDPROPERTY(loCust,"Company","West Wind Technologies")
ADDPROPERTY(loCust,"Entered",DATETIME())

*** Cursor as collection property of Customer obj
SELECT TOP 2 * FROM CUSTOMERS ORDER BY LastName INTO CURSOR TQuery
ADDPROPERTY(loCust,"CustomerList", "cursor:TQuery")

lcJson = loSer.Serialize(loCust)
```

Here the `.CustomerList` property is created as a property of the `loCust` object:

```
{
  "company": "West Wind Technologies",
  "customerlist": [
    {
      "id": "_4FG12Y7TK",
      "firstname": "Pat",
      "lastname": "@ Accounting",
      "company": "Windsurf Warehouse SF",
      "careof": "Pat @ Accounting",
      "address": "405 South Airport Blvd.\nSan Francisco, CA 94080",
      "entered": "2014-11-02T10:46:40Z",
      "state": "OR"
    },
    {
      "id": "_4FG12Y7U7",
      "firstname": "Steven",
      "lastname": "Black",
      "company": "SBC",
      "careof": "Steven Black",
      "address": "12 East Street\nKingston, ON\nK7K 6T3 Canada\n",
      "entered": "2014-06-02T09:46:40Z",
      "state": ""
    }
  ],
  "entered": "2021-09-25T04:49:02Z",
  "name": "Rick"
}
```

Field Casing and Name Overrides

One thing you might notice in all the examples above is that serialization causes all property names to be **lower case**. Most commonly JSON APIs return values in **camel case** which is lower case first word and capitalized sub-words. For example, a First Name field should be `firstName` using camel case.

Unfortunately FoxPro has no way to preserve case for field information in `AMEMBERS()` and `AFIELDS()` by default. Yeah, yeah I know you can use DBC field overrides or class name overrides, but these are not universally available and they don't work on things like `EMPTY` objects or properties added with `ADDPROPERTY()`.

So `wwJsonSerializer` provides an override for field names via the `cPropertyNameOverrides` which is provided as a comma delimited list of names like this:

```
loSer.PropertyNameOverrides = "firstName,lastName,customerList"
```

Note that I'm not naming all fields in this list - only the fields I actually need to override that have multipart names. With this in place the names are overridden in the JSON output here for the customer list embedded into an object from the last example:


```
{
  "company": "West Wind Technologies",
  "customerList": [
    {
      "id": "_4FG12Y7TK",
      "firstName": "Pat",
      "lastName": "@ Accounting",
      "company": "Windsurf Warehouse SF",
      "careof": "Pat @ Accounting",
      "address": "405 South Airport Blvd.\nSan Francisco, CA 94080",
      "entered": "2014-11-02T10:46:40Z",
      "state": "OR"
    },
    {
      "id": "_4FG12Y7U7",
      "firstName": "Steven",
      "lastName": "Black",
      "company": "SBC",
      "careof": "Steven Black",
      "address": "12 East Street\nKingston, ON\nK7K 6T3 Canada",
      "entered": "2014-06-02T09:46:40Z",
      "state": ""
    }
  ],
  "entered": "2021-09-25T04:49:02Z",
  "name": "Rick"
}
```

The `PropertyNameOverrides` property is immensely useful in ensuring that properties have the correct name. Since JSON is case sensitive, many services require that property names **match exactly** including case to update data.

Putting HTTP and JSON Together

At this point you have all the tools you need to:

- Serialize any data you need to send as JSON
- Call the server and send the data (if any)
- Get back a JSON Response
- Deserialize the JSON Response

So, let's do this, on a live service!

I'm going to use my [AlbumViewer sample](#) application on the West Wind Web Site that is publically accessible so we can play with the data. This happens to be a .NET API service, but I'll show you how to create a subset using a FoxPro service later in this article. For now we don't care **how** the data is created, only what shape it comes back as.

i HTTP == Technology Independence

Because REST is HTTP based, any type of application can access it. It doesn't matter whether the service was built with .NET, Java, Rust or Turtle Basic. All that matters is what the API output is in order to consume it.

You can also flip this concept around, and switch out the backend technology without affecting the client. So you can create the same interface in FoxPro or .NET. To access one or the other simply switch URLs. This can be a great migration path when updating to new technologies.

Compare that to something tech specific like COM or .NET or JAVA specific APIs which require platform specific tools/languages to interface with their respective APIs. With REST none of that matters because all we need is an HTTP client and a JSON serializer to access the data.

Retrieving a Collection of Simple Objects

Let's start with an album list. This request retrieves an array of album objects that looks like this:

```
[
  {
    "AlbumCount": 5,
    "Id": 25,
    "ArtistName": "AC/DC",
    "Description": "AC/DC's mammoth power chord roar became one of the most influential hard rock sounds of the '70s. In its own way...",
    "ImageUrl": "https://cps-static.rovicorp.com/3/JPG_400/MI0003/090/MI0003090436.jpg?partner=allrovi.com",
    "AmazonUrl": "http://www.amazon.com/AC-DC/e/B000AQU2YI/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=1412245004&sr=8-1&tag=westwindtechn-20&linkId=SSZOE52V3EG4M4SW"
  },
  {
    "AlbumCount": 3,
    "Id": 12,
    "ArtistName": "Accept",
    "Description": "With their brutal, simple riffs and aggressive, fast tempos, Accept were one of the top metal bands of the early '80s...",
    "ImageUrl": "https://cps-static.rovicorp.com/3/JPG_400/MI0001/389/MI0001389322.jpg?partner=allrovi.com",
    "AmazonUrl": "http://www.amazon.com/Accept/e/B000APZ8S4/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=1412245037&sr=8-3&tag=westwindtechn-20&linkId=KM4RZR3ECUXWBJ6E"
  },
  ...
]
```

To use this data in FoxPro we'll download the JSON and deserialize it using the following code:

```

DO wwhttp
DO wwJsonSerializer

loHttp = CREATEOBJECT("wwHttp")
loSer = CREATEOBJECT("wwJsonSerializer")

*** Retrieve JSON Artist Array from Server
lcJson = loHttp.Get("https://albumviewer.west-wind.com/api/artists")

*** Turn Array into FoxPro collection
loArtists = loSer.Deserialize(lcJson)

FOR EACH loArtist IN loArtists FoxObject
? loArtist.ArtistName + " (" + TRANSFORM(loArtist.AlbumCount) + ")"
ENDFOR

```

The `loHttp.Get()` call makes an HTTP `GET` request to retrieve data from the server. The captured JSON Array string is deserialized into a FoxPro collection and then displayed.

No rocket science here.

A more Complex Object

The artist list is a simple collection of a flat object, but the data can be more much more complex. For example, here's the code that retrieves a single artist, along with its related albums and tracks:

```

{
  "Artist": {
    "Id": 33,
    "ArtistName": "Anti-Trust",
    "Description": "Anti-Trust is a side project by ex-Attitude Adjustment members Chris Kontos, Rick Strahl and Andy Andersen. This collaboration produced....",
    "ImageUrl": "https://anti-trust.rocks/images/Photo6.jpg"
    "AmazonUrl": "https://anti-trust.rocks"
  },
  "Albums": [
    {
      "Id": 37,
      "Title": "Guilty",
      "Description": "Old school hardcore punk with metal roots, kicked out in good old garage style. Garage recorded by ex-Attitude Adjustment members Rick Strahl and Chris Kontos early in 2001-2002...",
      "Year": 2020,
      "ImageUrl": "https://anti-trust.rocks/Guilty-Cover.png",
      "AmazonUrl": "https://store.west-wind.com/product/order/antitrust_guilty",
      "SpotifyUrl": "https://anti-trust.rocks",
      "ArtistId": 33,
      "Tracks": [
        {
          "Id": 191,
          "AlbumId": 37,
          "SongName": "No Privacy",
          "Length": "2:22"
        },
        {
          "Id": 194,
          "AlbumId": 37,
          "SongName": "Anti-social",
          "Length": "2:25"
        },
        {
          "Id": 184,
          "AlbumId": 37,
          "SongName": "Fear Factory",
          "Length": "2:50"
        },
        ...
      ]
    }
  ]
}

```

This object is a 'container object' that contains two top level objects `Artist` and `Albums`. You can capture this structure in FoxPro easily. The code to retrieve and parse this JSON looks like this:

```
loHttp = CREATEOBJECT("wwhttp")
lcJson = loHttp.Get("https://albumviewer.west-wind.com/api/Artist/33")

loSer = CREATEOBJECT("wwJsonSerializer")
loArtist = loSer.Deserialize(lcJson)

? loArtist.Artist.ArtistName
? loArtist.Artist.Description

FOR EACH loAlbum in loArtist.Albums FOXOBJECT
  ? " -- " + loAlbum.Title + " (" + TRANSFORM(loAlbum.Year) + ")"
  FOR EACH loTrack IN loAlbum.Tracks FOXOBJECT
    ? "   -- " + loTrack.SongName
  ENDFOR
ENDFOR
```

As you can see it's quite easy to transport very complex structures over JSON back into a FoxPro object structure.

Updating an Object

Next let's look at **sending data** to the server in order to update an artist. To do this we'll need to create JSON, and send it to the server via a `POST` operation. The next thing we want to do is update an Artist.

It turns out that's actually a two step process:

- You need to authenticate to retrieve a Bearer Token
- Update the artist and provide the Bearer Token

So lets start with the authorization.

i Web Request Testing Tools

The first thing I recommend when you're working with APIs that have more than a few requests, is to use a URL testing tool to set up and play API requests separately from the application. This makes it easier to figure out exactly what you need to send to server and what it's sending back exactly.

A couple of common URL Testing tools are:

- **Postman**
- **West Wind WebSurge.**

Either of these tools let you create and save requests and then play them back to test requests and see the results. You can also share requests with others, so multiple users can work with the same test data. WebSurge can also do performance load testing on the URLs in a session.

Here's what WebSurge looks like with the request and response for the `Authenticate` request:

AlbumViewer.websurge - West Wind WebSurge (unregistered)

File Session Requests Help

Open Session | Run Url | Run All | Threads: 30 | Time: 200 secs | Export Results | Charts

Http Request | Preview

Request Headers

POST https://albumviewer.west-wind.com/api/authenticate HTTP/1.1

Accept: application/json
Accept-Encoding: gzip, deflate, br
User-Agent: West Wind HTTP .NET Client
Content-Type: application/json

Request Content ☒ Formatted json

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

Response Headers 435 bytes Time: 344ms First byte: 334ms

HTTP/1.1 200 OK

Server: Microsoft-IIS/10.0
X-Powered-By: ASP.NET
Date: Sun, 26 Sep 2021 04:37:34 GMT
Content-Length: 435
Content-Type: application/json; charset=utf-8

Response Content

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ0ZXN0IiwianRpIjoiaGR0cHM6Ly9hbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9",
  "expires": "2021-09-26T05:37:34Z",
  "displayName": "Test User"
}
```

Ready 0 Requests C:\projects\AlbumViewer\Next\AlbumViewer.websurge

This request requires that you send a username and password in an object and receive back a Token that can then be used in an `Authorization` header as a bearer token. I'll break this down into two sections, but the two operations happen in a single sequence. Here's the Authentication bit.

```

LOCAL loHttp as wwHttp, loSer as wwJsonSerializer
loSer = CREATEOBJECT("wwJsonSerializer")
loHttp = CREATEOBJECT("wwhttp")

*** Create the User Info object
loUser = CREATEOBJECT("EMPTY")
ADDPROPERTY(loUser,"Username", "test")
ADDPROPERTY(loUser, "Password", "test")
lcJson = loSer.Serialize(loUser)

*** We're sending JSON to the server and retrieve JSON back
loHttp.cContentType = "application/json"
lcJson = loHttp.Post("https://albumviewer.west-wind.com/api/Authenticate", lcJson)

*** Check for errors
IF loHttp.nError # 0
    ? "Failed: " + loHttp.cErrorMsg
ENDIF
IF loHttp.cResultCode = "401"
    ? "Login failed. Invalid credentials"
    RETURN
ENDIF
IF loHttp.cResultCode # "200"
    ? "Failed: " + loHttp.cResultCode + " " + loHttp.cResultCodeMessage
    RETURN
ENDIF

*** Deserialize the returned Object
loAuth = loSer.Deserialize(lcJson)

IF EMPTY(loAuth.Token)
    ? "Authentication failed. Invalid token."
    RETURN
ENDIF

lcToken = loAuth.Token && YAY!

```

Here I use a `POST` operation to send the username and password serialized from an object. Notice I added some basic error checking for failure of the HTTP request (if the connection can't be made or the server is down etc.) and checking the result code for the request. If auth fails the result code will be `401` and we have invalid credentials. The server actually returns an error message and we could peel that out of the data, but in this case the only failure is likely to be authentication failure (either way).

Ok, so now we have the token we need to use it with the follow-on request and pass it along with the updated (or new) Artist information to send to the server. Here's what the Artist request looks like in WebSurge (truncated data for brevity):

Request Headers

CopyRun

POST https://albumviewer.west-wind.com/api/artist HTTP/1.1

Accept: application/json
 Accept-Encoding: gzip, deflate
 Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
 User-Agent: West Wind HTTP .NET Client
 Content-Type: application/json

Request Content

Formatted json

```
{
  "Id": 33,
  "ArtistName": "Anti-Trust",
  "Description": "Anti-Trust is a side project by...",
  "ImageUrl": "https://anti-trust.rocks/images/Photo6.jpg",
  "AmazonUrl": "https://anti-trust.rocks"
}
```

Response Headers

2,298 bytesTime: 214msFirst byte: 214ms

HTTP/1.1 200 OK

Server: Microsoft-IIS/10.0
 X-Powered-By: ASP.NET
 Date: Sun, 26 Sep 2021 04:53:16 GMT
 Content-Type: application/json; charset=utf-8
 Content-Length: 2298

Response Content

```
{
  "Artist": {
    "Id": 33,
    "ArtistName": "Anti-Trust",
    "Description": "Anti-Trust is a side project by...",
    "ImageUrl": "https://anti-trust.rocks/images/Photo6.jpg",
    "AmazonUrl": "https://anti-trust.rocks"
  },
  "Albums": [
    {
      "Id": 37,
      "Title": "Guilty",
      "Description": "Old school hardcore punk with metal roots...",
      "Year": 2020,
      "ImageUrl": "https://anti-trust.rocks/Guilty-Cover.png",
      "AmazonUrl": "https://store.west-wind.com/product/order/antitrust_guilty",
      "SpotifyUrl": "https://anti-trust.rocks",
      "ArtistId": 33,
      "Tracks": [
        {
          "Id": 182,
          "AlbumId": 37,
          "SongName": "Protest the Possibilities (Discharge)",
          "Length": "3:10",
          "Bytes": 0,
          "UnitPrice": 0.0
        }
      ]
    }
  ]
}
```


You can see that we send a simple flat artist object, which updates matching properties on the server. The server then returns a **fully populated Artist object** which includes related albums.

```
*** Our token from the code above - continuing on
lcToken = loAuth.Token

*** Create an Artist object - could also come from cursor SCATTER NAME etc.
loArtist = CREATEOBJECT("EMPTY")
ADDPROPERTY(loArtist, "Id", 33)
ADDPROPERTY(loArtist, "ArtistName", "Anti-Trust")
ADDPROPERTY(loArtist, "Description",;
    "Anti-Trust is a side project by ex-Attitude Adjustment members " +;
    "Chris Kontos, Rick Strahl and Andy Andersen. This collaboration " +;
    "produced a handful of songs that were garage recorded in " +;
    "Oakland, CA and Maui, HI in 2001 and 2002 by Rick and Chris. " +;
    "Several additional songs were recorded in late 2020 and early 2021 " +;
    "which resulted in a the songs being officially put out and released " +;
    "online and in album form." + CHR(10) + CHR(10) + ;
    "Anti-Trust's music features diverse influences from old school hardcore punk, " +;
    "metal cross over and NWOFBHM, all driven by heavy guitar rhythms " +;
    "and catchy choruses with a unique and edgy sound.")
ADDPROPERTY(loArtist, "ImageUrl", "https://anti-trust.rocks/images/Photo6.jpg")
ADDPROPERTY(loArtist, "AmazonUrl", "https://amzn.to/3ucZIPk")
ADDPROPERTY(loArtist, "SpotifyUrl", "https://anti-trust.rocks")

lcJson = loSer.Serialize(loArtist)

*** Now add the Token in Bearer Authentication
loHttp.AddHeader("Authorization", "Bearer " + lcToken)

*** Must specify we're sending JSON
loHttp.cContentType = "application/json"

*** Update existing record with POST or PUT
lcJson = loHttp.Post("https://albumviewer.west-wind.com/api/Artist", lcJson)

*** Error Handling
IF loHttp.nError # 0
    ? "Failed: " + loHttp.cErrorMsg
ENDIF
IF loHttp.cResultCode # "200"
    ? "Failed: " + loHttp.cResultCode + " " + loHttp.cResultCodeMessage
    RETURN
ENDIF

*** Retrieve artist object from server (overwrites old object!)
```

```

loArtist = loSer.Deserialize(lcJson)

*** for new records we might want to know the new id
InId = loArtist.Id

*** Just for (not very practical) kicks print out Artist, Albums, Tracks
? loArtist.Artist.ArtistName
? loArtist.Artist.Description

FOR EACH loAlbum in loArtist.Albums FOXOBJECT
  ? " -- " + loAlbum.Title + " (" + TRANSFORM(loAlbum.Year) + ")"
  FOR EACH loTrack IN loAlbum.Tracks FOXOBJECT
    ? "  -- " + loTrack.SongName
  ENDFOR
ENDFOR

```

Again - this should all look pretty familiar by now - the process is the same: Take an object to send and serialize into JSON, send it, retrieve the result, check for errors, deserialize from JSON. Rinse and repeat for other requests. The structure can be much deeper.

In this example (`PostArtist.prg`) I do both the authentication and artist update in the same bit of code. Realistically you'd want to separate the Authentication code into an easily reusable function/method that you can call more easily. Also, if you're consuming this data, you'd likely call Authenticate once and then cache the Token in a global variable or other state, and simply reuse it.

Deleting an Object

Deleting an object is as simple as using the `DELETE` HTTP verb on the `/api/artist` URL. Note that the URL is overloaded for `POST`, `PUT` and `DELETE` operations which have different behavior even though they point at the same URL.

The delete operation looks like this:

Request Headers

Copy
 Run

```

DELETE https://albumviewer.west-wind.com/api/artist/29 HTTP/1.1

Accept-Encoding: gzip,deflate
Accept: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
  .eyJzdWIiOiJ0ZXN0IiwianRpIjoiodRiYWMyYtZTZhZS00NzQ5LWJkNjItYzZmMGM4ZmZhZjk5IiwiaXN1c1N0YXR1IjoivGVzdCB
  Vc2VyfHRlc3R8FEZhbHN1fDF8bnVsbHxkZDkxNTM0MyIsImV4cCI6MTYzMjYzNTQ1MSwiaXNzIjoiaHR0cHM6Ly9hbGJ1bXZpZXdlci5
  3ZXN0LXdpbmQuY29tIiwiaXVkiIjoiaHR0cHM6Ly9hbGJ1bXZpZXdlci53ZXN0LXdpbmQuY29tIn0
  .31kheFnnjYBaJ1CA7dyjkNVluk6iv_m8dyDvREdpdJ8
User-Agent: West Wind HTTP .NET Client

```

Response Headers

4 bytes Time: 126ms First byte: 126ms

```

HTTP/1.1 200 OK

Server: Microsoft-IIS/10.0
X-Powered-By: ASP.NET
Date: Sun, 26 Sep 2021 05:08:34 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 4

```

Response Content

```
true
```

I'm not going to show a code example here since this code won't work repeatedly as items disappear once deleted. The key feature is to use `loHttp.Delete(lcUrl)` to execute the request. In this case the API simply returns a single boolean value of `true` or `false`. Actually it'll always return `true` or an error response.

I'll talk more about error handling when we look at Server code later on in this article.

Removing Repetitive Code with `wwJsonServiceClient`

If you look at the above examples you're probably noticing that a lot of that code is repeated over and over. Creating a serializer, and HTTP object, setting up the data to send and receive, checking for errors etc. There's a lot of boilerplate code in there that can actually be abstracted away.

If using West Wind Tools you can use the `wwJsonService class` which is basically a JSON client that combines the features of `wwHttp` and `wwJsonSerializer` into a simpler abstraction. The service client abstracts and handles:

- JSON Serialization in and out
- The HTTP Call
- Error Handling

It basically lets you boil a REST client call down to a single line of code, plus configuration (if any). So rather than manually serializing, you pass in your raw FoxPro values, objects, cursors and the client makes the HTTP call and returns the deserialized result back to you as a FoxPro value, object or collection. If something goes wrong, the client provides a simple way to check for errors using `lError` and `cErrorMsg` properties.

You can use the service in two ways:

- Directly as a generic REST Service client

- Subclassed as a Service Wrapper Class

Generic REST Service Client

The raw service client class can be used to make calls against a service directly. You use the `CallService()` method to provide inputs and it goes out and makes the call and returns the result, all using standard FoxPro values, objects, collections and cursors.

Let's do the simple Artist list retrieval first:

```
loClient = CREATEOBJECT("wwJsonServiceClient")
loArtists = loClient.CallService("https://albumviewer.west-wind.com/api/artists")

FOR EACH loArtist IN loArtists FoxObject
? loArtist.ArtistName + " (" + TRANSFORM(loArtist.AlbumCount) + ")"
ENDFOR
```

Simple right? This defaults to a `GET` request against the server with no data sent.

To demonstrate sending data, let's review the previous dual request Artist update example. If you recall in that example I first authenticated then sent the updated Artist to server via a `POST`. Here's that code with the service client:

```
*** Create an object
loUser = CREATEOBJECT("EMPTY")
ADDPROPERTY(loUser,"Username", "test")
ADDPROPERTY(loUser, "Password", "test")

loClient = CREATEOBJECT("wwJsonServiceClient")

*** Pass the object for POST and return Auth Object
loAuth = loClient.Callservice("https://albumviewer.west-wind.com/api/Authenticate", loUser, "POST")

IF loClient.IError
? "Failed: " + loClient.cErrorMsg
RETURN
ENDIF

*** Yay we got a token!
lcToken = loAuth.Token
IF EMPTY(lcToken)
? "Authentication failed. Invalid token."
RETURN
ENDIF

loArtist = CREATEOBJECT("EMPTY")
ADDPROPERTY(loArtist, "Id", 33)
ADDPROPERTY(loArtist, "ArtistName", "Anti-Trust")
```

```

... && more ADDPROPERTY() calls as before

*** Create clean client instance
loClient = CREATEOBJECT("wwJsonServiceClient")
loClient.oHttp.AddHeader("Authorization", "Bearer " + lcToken)

*** Pass loArtist directly get updated Artist instance
loUpdated = loClient.CallService("https://albumviewer.west-wind.com/api/Artist", loArtist, "POST")

IF loClient.IError
    ? "Failed to update: " + loClient.cErrorMsg
    RETURN
ENDIF

? loUpdated.Artist.ArtistName
? loUpdated.Artist.Description

FOR EACH loAlbum IN loUpdated.Albums FOXOBJECT
    ? " -- " + loAlbum.Title + " (" + TRANSFORM(loAlbum.Year) + ")"
    FOR EACH loTrack IN loAlbum.Tracks FOXOBJECT
        ? "   -- " + loTrack.SongName
    ENDFOR
ENDFOR
ENDFOR

```

The key pieces here are the two `CallService()` calls that call the server **with data**. This one sends the auth information and returns a server auth object with a token on success:

```
loAuth = loClient.CallService("https://albumviewer.west-wind.com/api/Authenticate", loUser, "POST")
```

Notice that you specify a raw FoxPro object (`loUser`) and specify the HTTP Verb and the object (or value, collection or cursor using `cursor:TUser` syntax) to send to the server. No explicit serialization required. As with the Auth request, the result also comes back as a FoxPro object, that can be walked through and in this case displayed.

Creating a REST Service Client SubClass

The generic `wwJsonServiceClient` works great to reduce busy work when making service calls, but I recommend taking this one step further by creating specific Service classes that **inherit from `wwJsonServiceClient`** in order to provide a business level abstraction, similar to a business object.

So rather than using `wwJsonServiceClient` directly you subclass and then create methods for each service call using the methods of this class. Given the examples I've shown here we might have methods like:

- `GetArtists()`
- `GetArtist(InId)`
- `UpdateArtist(loArtist)`
- `DeleteArtist()`

I'm going to start out with the two `GET` operation because they are the simplest:

```
DO wwhttp
DO wwJsonSerializer
SET PROCEDURE TO artistService ADDITIVE

*****

DEFINE CLASS ArtistService AS wwJsonServiceClient
*****

*** Always abstract the base path so you can switch sites
*** easily. Useful for debugging, local, live, staging etc.
cServiceBaseUrl = "https://albumviewer.west-wind.com/"

*****

* GetArtists
*****

FUNCTION GetArtists()
LOCAL loArtists

loArtists = this.CallService( this.cServiceBaseUrl + "api/artists")
IF this.IError
    RETURN NULL
ENDIF

RETURN loArtists
ENDFUNC

* GetArtists

*****

* GetArtist
*****

FUNCTION GetArtist(InId)
LOCAL loArtist

loArtist = this.CallService( this.cServiceBaseUrl + "api/artist/" + TRANSFORM(InId) )
IF THIS.IError
    RETURN NULL
ENDIF

RETURN loArtist
ENDFUNC

* GetArtist

ENDDEFINE
```

I start by subclassing `WwJsonServiceClient` and adding a `cServiceBaseUrl` property. I highly recommend to **never hardcode server paths** because it's almost certain that you will need to switch servers at some point. Whether it's for dev vs. live or staging, or because you're changing to a new server or adding a second domain. Never hardcode server paths.

The actual service methods then tend to be super simple delegating most of the work to the `CallService` method. You can do more in these methods if you want - like validate incoming data, or combine multiple service calls into single methods. More on these in a minute.

But one thing that you always want to do is provide application specific error handling. Personally I like to handle errors in my operations and return a value from the function that's easy to check. When returning objects, a failure typically ends up returning null. For strings, perhaps an empty string (or NULL) etc. This makes it more natural to check for errors with just a return value.

To use the `ArtistService` is now a piece of cake with code that at the application level doesn't directly interact with HTTP or JSON or even a service client. For all intents and purposes this code looks more like calling a business object:

```
DO ArtistService
```

```
LOCAL loService as AristService
```

```
loService = CREATEOBJECT("ArtistService")
```

```
CLEAR
```

```
? "*** ARTIST LIST"
```

```
?
```

```
loArtists = loService.GetArtists()
```

```
FOR EACH loArtist IN loArtists FoxObject
```

```
? loArtist.ArtistName + " (" + TRANSFORM(loArtist.AlbumCount) + ")"
```

```
ENDFOR
```

```
WAIT WINDOW TIMEOUT 10
```

```
CLEAR
```

```
? "*** SINGLE ARTIST"
```

```
?
```

```
loArtist = loService.GetArtist(33)
```

```
? loArtist.Artist.ArtistName
```

```
? PADR(loArtist.Artist.Description,1000)
```

```
FOR EACH loAlbum in loArtist.Albums FOXOBJECT
```

```
? " -- " + loAlbum.Title + " (" + TRANSFORM(loAlbum.Year) + ")"
```

```
FOR EACH loTrack IN loAlbum.Tracks FOXOBJECT
```

```
? " -- " + loTrack.SongName
```

```
ENDFOR
```

```
ENDFOR
```

```
WAIT WINDOW TIMEOUT 10
```

Next lets look at `Authenticate` and `UpdateArtist`. To make things a little more interesting I'll add a little more logic to these to make them more useful here rather than just exposing the service calls. I'll aggregate `Authenticate()` inside of `UpdateArtist()` and provide rudimentary auto-authentication.

I'm going to add three more properties to the class:


```
DEFINE CLASS ArtistService AS wwJsonServiceClient
```

```
cServiceBaseUrl = ""
```

```
cBearerToken = ""
```

```
cUsername = ""
```

```
cPassword = ""
```

```
FUNCTION Init(lcBaseUrl, lcUsername, lcPassword)
```

```
IF !EMPTY(lcBaseUrl)
```

```
    this.cServiceBaseUrl = RTRIM(lcBaseUrl, "/") + "/"
```

```
ENDIF
```

```
IF !EMPTY(lcUserName)
```

```
    this.cUsername = lcUsername
```

```
ENDIF
```

```
IF !EMPTY(lcPassword)
```

```
    this.cPassword = lcPassword
```

```
ENDIF
```

```
ENDFUNC
```

```
* Init
```

And then add the Authenticate method. Rather than return the raw service result, a success request sets the `cBearerToken` property which can then be used on later requests.

```

*****
FUNCTION Authenticate(lcUsername, lcPassword)
*****

IF EMPTY(lcUsername) AND EMPTY(lcPassword)
    lcUsername = this.cUsername
    lcPassword = this.cPassword
ENDIF
IF EMPTY(lcUsername) AND EMPTY(lcPassword)
    this.cErrorMsg = "Username and password cannot be empty."
    RETURN .F.
ENDIF

loLogin = CREATEOBJECT("EMPTY")
ADDPROPERTY(loLogin, "userName", lcUsername)
ADDPROPERTY(loLogin, "password", lcPassword)

loAuth = this.CallService(this.cServiceBaseUrl + "api/authenticate",loLogin,"POST")
IF this.IError
    RETURN .F.
ENDIF

THIS.cBearerToken = loAuth.Token
RETURN .T.
ENDFUNC
* Authenticate

```

This method shows why it can be useful to abstract service functionality into a class as you can add additional wrapping logic to the service call logic. Here the input data is validated prior to calling the service method. Also notice than rather than requiring an object to be passed in, I simply use parameters to create an object on the fly to use for the service call.

Then `CallService()` is used and the on the fly created `loLogin` record is posted to the service. If the call succeeds the `.cBearerToken` property is set with the returned token value and the method returns `.T.` If validation or the service call fail `.F.` is returned.

In short, this method signature looks very different than the underlying service call, and provides some additional functionality that the service call alone does not have.

To build on this logic, the `UpdateArtist()` can then actually use `Authenticate()` as part of its logic:

```

*****
FUNCTION UpdateArtist(loArtist)
*****

LOCAL loUpdated

IF EMPTY(THIS.cBearerToken)
    IF !this.Authenticate()
        RETURN NULL
    ENDIF
ENDIF

IF THIS.IError
    RETURN NULL
ENDIF

*** Add the auth header
THIS.oHttp.Addheader("Authorization", "Bearer " + this.cBearerToken)
loUpdated = THIS.CallService( THIS.cServiceBaseUrl + "api/artist", loArtist, "POST")
IF this.IError
    RETURN NULL
ENDIF

RETURN loUpdated
ENDFUNC
* UpdateArtist

```

Notice the first block of code that checks the `cBearerToken` and if not set calls to `Authenticate()`. If authentication fails the error from it's failure will show up as the error message. If the Update fails it will get its own error message.

The update service call is then just another - by now boring - `CallService()` call that posts an object to the server. Easy peasy.

To use this method becomes now very simple:

```

? "*** UPDATE ARTIST"
?

*** Create service and pass url, uid/pwd
loService = CREATEOBJECT("ArtistService","", "test", "test")

loArtist = CREATEOBJECT("EMPTY")
ADDPROPERTY(loArtist, "Id", 33)
ADDPROPERTY(loArtist, "ArtistName", "Anti-Trust")
ADDPROPERTY(loArtist, "Description", ;
    "UPDATED! Anti-Trust is a side project by ex-Attitude Adjustment members " +;
    "...")
ADDPROPERTY(...)

*** Return new artist object from server
loArtist = loService.UpdateArtist(loArtist)

IF ISNULL(loArtist)
    ? loService.cErrorMsg
    RETURN
ENDIF

? loArtist.Artist.ArtistName
? loArtist.Artist.Description  && updated value here

```

Most of this code should look familiar from previous examples, but the key bits of this code are these two lines:

```

loService = CREATEOBJECT("ArtistService","", "test", "test")
loArtist = loService.UpdateArtist(loArtist)

```

We've come a long way from manually running HTTP requests and serializing and parsing JSON to making a simple business object like method call!

The code above handles both authentication and the artist update as part of a single operation. This is what aggregation is all about, and it lets you compose complex functionality from relatively simple service calls into coordinated logic that is handled in a central and easily maintainable, business-object-like class.

Whether you use `wwJsonServiceClient` or not, **I highly recommend** some mechanism like this for isolating your application code from the underlying service handling. Wrapper methods like this let your application use a natural interface, and push all the system level gunk either in the framework (via `wwJsonServiceClient` or if doing it by hand in the actual method code. This makes the code more re-usable, more maintainable and also in the future replaceable should you decide to change services.

Creating a REST Service with Web Connection

So far I've shown how to consume a REST service from Visual FoxPro using an online .NET based service and that works just fine. Clients consuming a Web API couldn't care less what technology the API is written in.

That's one of the benefits of exposing functionality as a service in the first place.

Which brings us to the next topic, which is to talk about how to create a REST Web API using Visual FoxPro. Unlike SOAP Services which were super complex to set up and manage as they had to support all of the complex SOAP protocol parsing features, REST Services using JSON are very simple and can be implemented even manually with any Web framework. If you're using specialized tools, they likely have support for creating service in a more natural way such as an MVC framework where the `View` is the JSON output.

In this article I use West Wind Web Connection which has both manual JSON support as part of the core framework, or native support for Web APIs via a special `wwProcess` class called `wwRestProcess`. You use either. Manual support is very similar to the code we used

Manual JSON Handling in Web Connection

Web Connection has rich support for REST services via a custom REST Service class, but if you have an old application and you have one or two API requests you need to serve you probably don't want to add a new process class to an existing project.

Manual processing is easy to do, but requires a few steps:

- Capture the `Request.Form()` content for JSON inputs
- Deserialize JSON to a FoxPro object/collection/value
- Do your normal request processing
- Create result in the form of a FoxPro Object or Cursor
- Serialize result JSON

Keeping with the MusicStore examples I showed for the client let's accept a customer update request to demonstrate both receiving and sending of JSON data in a Web request:

```

FUNCTION UpdateArtist()
LOCAL lcJson, loSer, loArtistBus, loArtist

lcJson = Request.Form() && Retrieve raw POST buffer

*** Deserialize the incoming JSON text
loSer = CREATEOBJECT("wwJsonSerializer")
loArtistEntity = loSer.Deserialize(lcJson)

*** Do our business logic using a Business object
loArtistBus = CREATEOBJECT("cArtist")

*** Load an entity into .oData property
IF !loArtistBus.Load(loArtistEntity.Id)
    loArtistBus.New()
ENDIF

loArtist = loArtistBus.oData && easier reference

*** update loaded data
loArtist.ArtistName = loArtistEntity.ArtistName
loArtist.Description = loArtistEntity.Description
...

IF !loArtistBus.Save()
    *** Always try to return JSON including for errors
    RETURN "{ isError: true, message: 'Couldn't save customer.' }";
ENDIF

lcJson = loSer.Serialize(loArtistBus.oData)

*** Write the output into the Response stream as JSON
Response.ContentType = "application/json"
Response.Write(lcJson)

```

This code should be pretty self explanatory. Web Connection routes the request along with the JSON payload, which the method picks up and processes. The end result is an object, that is then serialized and pushed out via the Response.

Using a Web Connection REST Service Process

If you are building a Web application that is based around APIs or services you'll want to separate out your API Service into a separate class. Web Connection includes a custom `wwRestProcess` class that provides for standard Web Connection process that:

- Routes requests to a Process Method
- Deserializes a single JSON object/value and passes it as an Input Parameter

- Serializes a single value returned into JSON

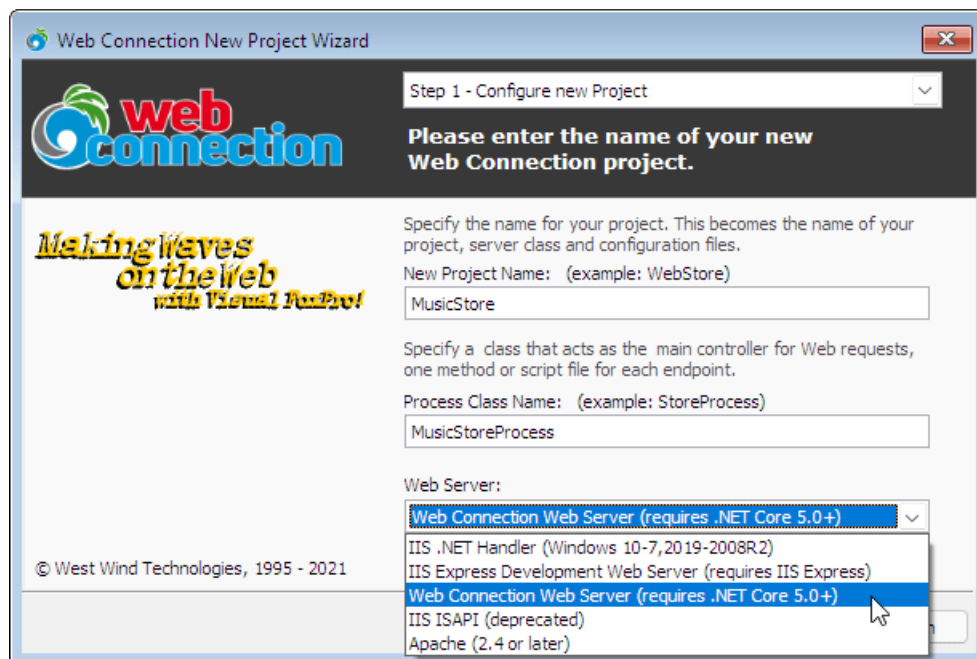
This class preserves all of Web Connection's functionality except it modifies how input is provided to the method and how output is returned. The biggest difference is that typically with `wwRestProcess` you don't use the `Response` object to send output, but rather just return a value.

Let's take a look at a setting up a REST service with Web Connection.

Creating a new REST API Service with Web Connection

The easiest way to create a new REST Service is to use the New Project (or new Process) Wizard which creates a new application for you. Start the Web Connection Console with `DO CONSOLE` and choose **New Project**.

First we'll create the new project called `MusicStore` with a process class called `MusicStoreProcess` and specify the **Web Server we want to use locally**:



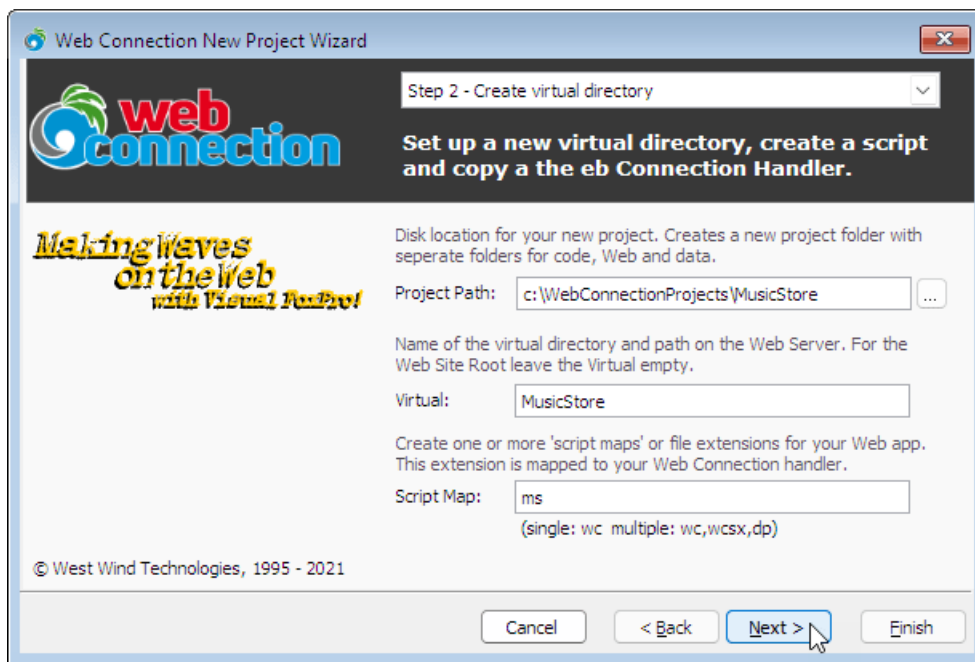
*Note all Web Servers require either **configuration or installation of some tools***

I'm using the local .NET Core based Web Connection Web Server here.

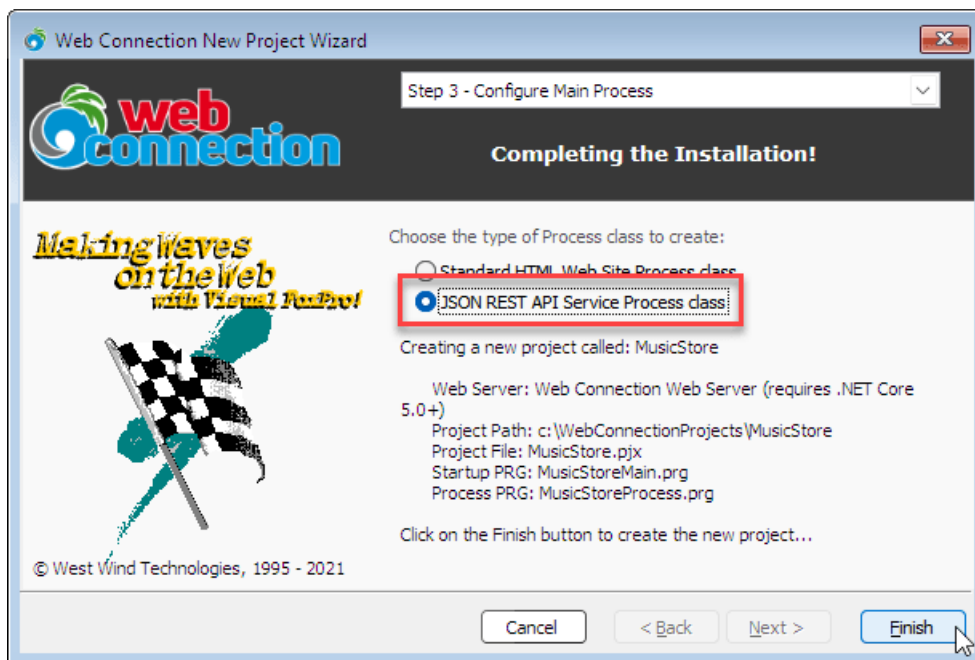
Next we get to specify where to create the project folder, and what virtual (if any and for IIS only) we want to use.

Web Connection creates a self-contained project folder which contains the Web files (Web), the FoxPro source files (Deploy) and an optional Data folder. The structure is self contained and fixed, so that the entire project can be easily moved and automatically be configured in a new location where all system files and folders are relative to each other in known locations.

We also need to specify a scriptmap - an extension for 'pages' that we access (ie. `Artist.ms`) - that tells IIS to route any request URLs with that extension (`.ms`) to our Process class:



Finally we need to specify that we want to create a **JSON REST API Service Process class** rather than a standard Web Connection process class for HTML applications.



When it's all said and done Web Connection launches:

- The Web Server (for IIS Express and Web Connection Web Server)
- The Web Connection Server (ie. `MusicStoreMain.prg`)
- A Web Browser to the root Web Site (`https://localhost:5200`)

If you click the **Hello World** link on the sample site you should now see a JSON result returned from the `TestPage.ms` request.

Here's what all that looks like:

The screenshot displays a development environment with three main components:

- Command Prompt (Top Left):** Shows the execution of `WebConnectionWebServer v7.22.1`. It lists configuration details such as Site Url (`http://localhost:5200`), Web Root (`c:\webconnectionprojects\musicstore\web`), and Environment (`Production`). Below this, a list of HTTP GET requests to `localhost:5200` is shown with their respective response times (e.g., `200 10ms` for `/Default.htm`).
- Web Browser (Top Right):** Displays the `MusicStore Test Page` at `localhost:5200`. The page features a "Web Connection" header, "Feature Samples", and "Application Links" including "Hello World Test Page", "Hello Scripting World", and "Web Connection Administration".
- Code Editor (Bottom):** Shows a file named `MusicStore - West Wind Web Connection`. The code includes instructions for launching the web server, setting web servers to "IIS" or "Fox Server", and launching the web URL (`http://localhost:5200`). A small inset window titled "West Wind Web Connection" shows a log of requests:


```

12:10:09 /testpage.ms - 0.096
12:10:11 /HelloScript.ms - 0.026
12:10:16 /testpage.ms - 0.003
      
```

If you click on the **Hello World** request and the scripting link in this REST Service you get back JSON responses. Here's the Hello World response:

```

{
  Description: "This is a JSON API method that returns an object.",
  Entered: "2021-09-27T19:12:43Z",
  Name: "TestPage"
}
  
```

This is generated inside of the `MusicStoreProcess.prg` class where there is a method called `TestPage` that

looks like this:

```
*****  
FUNCTION TestPage  
*****  
LPARAMETERS lvParm  && any posted JSON object (not used here)  
  
*** Simply create objects, collections, values and return them  
*** they are automatically serialized to JSON  
loObject = CREATEOBJECT("EMPTY")  
ADDPROPERTY(loObject,"name","TestPage")  
ADDPROPERTY(loObject,"description",;  
            "This is a JSON API method that returns an object.")  
ADDPROPERTY(loObject,"entered",DATETIME())  
  
*** To get proper case you have to override property names  
*** otherwise all properties are serialized as lower case in JSON  
Serializer.PropertyNameOverrides = "Name,Description,Entered"  
  
RETURN loObject
```

This simple method demonstrates the basics of how REST Endpoints work in Web Connection:

- Single input parameter for a JSON Object POSTed (if POST/PUT)
- Method body that creates a result object or cursor
- RETURN a plain FoxPro object or Cursor (`cursor:TCompany`)

Pretty simple right?

Creating the API Artist EndPoints

Ok let's dive in then and create the service interface for:

- Retrieving an Artist List
- Retrieving an individual Artist
- Updating an Artist
- Deleting an Artist

Returning a list of Artists from a Cursor

The first request will be the Artist list that is returned as a cursor.

To create a new endpoint method in Web Connection all we need to do is add another method to the `MusicStoreProcess` class. I'm going to use a business object class for the Artist operations that work against a data set. You can find both of these with the sample data on [GitHub](#).

- [MusicStore Business Objects](#)
- [MusicStore Data Files](#)

Here's the Artists method which can be accessed with `http://localhost:52000/Artists.ms` :

```

*****
FUNCTION Artists()
*****

loArtistBus = CREATEOBJECT("cArtist")
lnArtistCount = loArtistBus.GetArtistList()

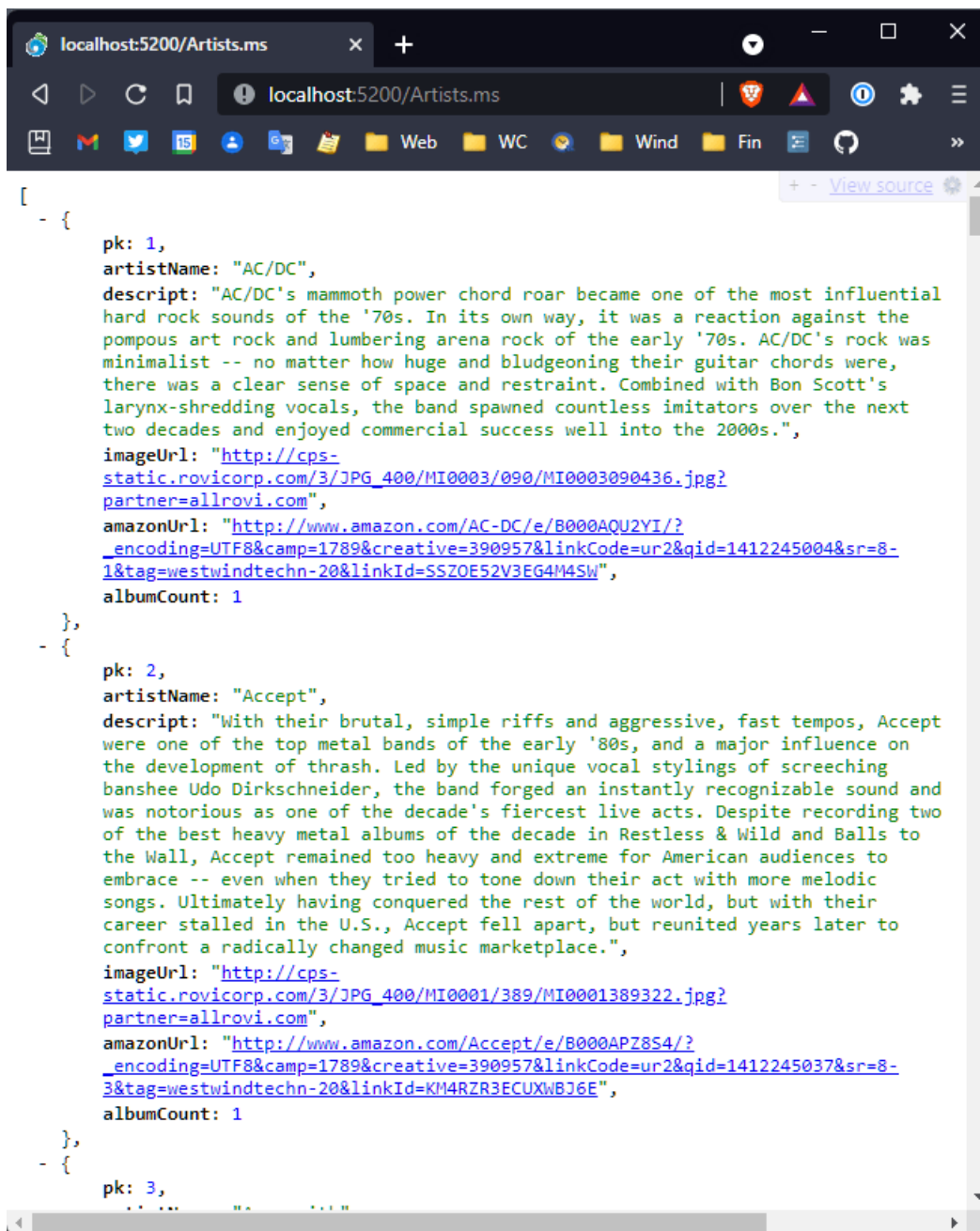
Serializer.PropertyNameOverrides = "artistName,imageUrl,amazonUrl,albumCount,"

RETURN "cursor:TArtists"
ENDFUNC

```

The code for this bit is very simple: The business object returns a list of all articles as a cursor named `TArtists` and we return that cursor as a result of the method via the same `cursor:TArtists` syntax that we used earlier when generating JSON on the client. No surprise there - the server framework is using the same serializer.

You can open this URL in the browser and if you have a [JSON addin](#) you can see nicely formatted JSON:



```
[
  - {
    pk: 1,
    artistName: "AC/DC",
    describe: "AC/DC's mammoth power chord roar became one of the most influential hard rock sounds of the '70s. In its own way, it was a reaction against the pompous art rock and lumbering arena rock of the early '70s. AC/DC's rock was minimalist -- no matter how huge and bludgeoning their guitar chords were, there was a clear sense of space and restraint. Combined with Bon Scott's larynx-shredding vocals, the band spawned countless imitators over the next two decades and enjoyed commercial success well into the 2000s.",
    imageUrl: "http://cps-static.rovicorp.com/3/JPG_400/MI0003/090/MI0003090436.jpg?partner=allrovi.com",
    amazonUrl: "http://www.amazon.com/AC-DC/e/B000AQU2YI/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=1412245004&sr=8-1&tag=westwindtechn-20&linkId=SSZOE52V3EG4M4SW",
    albumCount: 1
  },
  - {
    pk: 2,
    artistName: "Accept",
    describe: "With their brutal, simple riffs and aggressive, fast tempos, Accept were one of the top metal bands of the early '80s, and a major influence on the development of thrash. Led by the unique vocal stylings of screeching banshee Udo Dirkschneider, the band forged an instantly recognizable sound and was notorious as one of the decade's fiercest live acts. Despite recording two of the best heavy metal albums of the decade in Restless & Wild and Balls to the Wall, Accept remained too heavy and extreme for American audiences to embrace -- even when they tried to tone down their act with more melodic songs. Ultimately having conquered the rest of the world, but with their career stalled in the U.S., Accept fell apart, but reunited years later to confront a radically changed music marketplace.",
    imageUrl: "http://cps-static.rovicorp.com/3/JPG_400/MI0001/389/MI0001389322.jpg?partner=allrovi.com",
    amazonUrl: "http://www.amazon.com/Accept/e/B000APZ854/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=1412245037&sr=8-3&tag=westwindtechn-20&linkId=KM4RZR3ECUXWBJ6E",
    albumCount: 1
  },
  - {
    pk: 3,
```

Notice that property names are returned in *camelCase*. By default FoxPro will serialize only lower case property names, but because I used the `PropertyNameOverrides` property I can explicitly specify field names with custom case.

While the browser works for looking at `GET` requests, I prefer to set up URLs for testing in a separate tool like [Postman](#) or [WebSurge](#). This is especially useful if you need to `POST` data to the server since there's no easy way to do that repeatedly in the browser without creating a small app. Storing all the requests in one place is also a nice way to quickly see what operations are available on your API.

Here's the `Artists` request - and the others we'll create - in [WebSurge](#):

The screenshot shows the West Wind WebSurge application. The sidebar on the left contains a list of endpoints: Artist List, Get single artist, Update Artist, Delete Artist, Albums List, and Single Album. The 'Artist List' endpoint is selected, showing a GET request to /Artists.ms. The main area displays the request details, including the URL, HTTP method (GET), and headers. The response section shows the response headers and the response content, which is a JSON array of two artist objects.

```

GET http://localhost:5200/Artists.ms HTTP/1.1

Accept-Encoding: gzip, deflate
Accept: application/json
User-Agent: West Wind HTTP .NET Client

Response Headers
63,944 bytes Time: 136ms First byte: 134ms

HTTP/1.1 200 OK

Date: Mon, 27 Sep 2021 20:22:39 GMT
Server: Kestrel
Transfer-Encoding: chunked
RequestId: 21_2bb7841d
Content-Type: application/json; charset=utf-8

Response Content
[
  {
    "pk": 1,
    "artistName": "AC/DC",
    "descript": "AC/DC's mammoth power chord roar became one of the most influential hard rock sounds of the '70s. In its own way, it was a reaction against the pompous art rock and lumbering arena rock of the early '70s. AC/DC's rock was minimalist -- no matter how huge and bludgeoning their guitar chords were, there was a clear sense of space and restraint. Combined with Bon Scott's larynx-shredding vocals, the band spawned countless imitators over the next two decades and enjoyed commercial success well into the 2000s.",
    "imageUrl": "http://cps-static.rovicorp.com/3/JPG_400/MI0003/090/MI0003090436.jpg?partner=allrovi.com",
    "amazonUrl": "http://www.amazon.com/AC-DC/e/B000AQU2YI/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=1412245004&sr=8-1&tag=westwindtechn-20&linkId=SSZ0E52V3EG4M4SW",
    "albumCount": 1
  },
  {
    "pk": 2,
    "artistName": "Accept",
    "descript": "With their brutal, simple riffs and aggressive, fast tempos,"
  }
]

```

At the bottom of the application, a status bar indicates the current URL is http://localhost:5200/Artists.ms and shows 0 requests.

Object Composition: Retrieving an individual Artist

The previous request was a simple list result with flat objects. But you can also return much more complex structures that nest multiple objects and collections to create a

Returning an Artist returns a nested structure with Artist, albums and tracks. Lets see how this works.

The key to this is to use the business object to retrieve the base data and then composing a more complex object. Here's the Album method which responds to a URL like `https://localhost/artist?id=1`:

```

*****
FUNCTION Artist(loArtist)
*****

LOCAL InId, lcVerb, loArtistBus

InId = VAL(Request.QueryString("id"))
lcVerb = Request.GetHttpVerb()

if (lcVerb == "POST" or lcVerb == "PUT")
    RETURN this.UpdateArtist(loArtist)
ENDIF

IF lcVerb = "DELETE"
    loArtistBus = CREATEOBJECT("cArtist")
    RETURN loArtistBus.Delete(InId) && .T. or .F.
ENDIF

*** GET Operation
IF InId == 0
    RETURN this.ErrorResponse("Invalid Artist Id","404 Not Found")
ENDIF

loArtistBus = CREATEOBJECT("cArtist")
IF !loArtistBus.Load(InId)
    RETURN this.ErrorResponse("Artist not found. ","404 Not Found")
ENDIF

*** Lazy load the albums
loArtistBus.LoadAlbums()

Serializer.PropertyNameOverrides = "artistName,imageUrl,amazonUrl,albumCount,albumPk,
artistPk,songName,unitPrice,"

return loArtistBus.oData
ENDFUNC

```

The result is a complex object that returns an album top level object with a contained `albums` collection, each of which in turn has a `tracks` collection:

```

{
  "pk": 2,
  "artistName": "Accept",
  "descript": "With their brutal, simple riffs and aggressive...",
  "amazonUrl": "http://www.amazon.com/Accept/e/B000APZ8S4&linkId=KM4RZR3ECUXWBJ6E",
  "imageUrl": "http://cps-static.rovicorp.com/3/JPG_400/MI0001/389/M389322.jpg?partner=allrovi.com",
  "albums": [

```

```

{
  "amazonUrl": "http://www.amazon.com/gp/product/B00005NNMJ/&linkId=MQIHT543FNE5PNZU",
  "artist": {
    "albums": null,
    "amazonUrl": "http://www.amazon.com/Accept/e/B000APZ8S4/&linkId=KM4RZR3ECUXWBJ6E",
    "artistName": "Accept",
    "descript": "With their brutal, simple riffs and aggressive...",
    "imageUrl": "http://cps-static.rovicorp.com/3/JPG_40/MI01/389/M389322.jpg?partner=allrovi.com",
    "pk": 2
  },
  "artistPk": 2,
  "descript": "As cheesy as some of the titles and lyrics on this record are...",
  "imageUrl": "https://images-na.ssl-images-amazon.com/images/I/519J0xGWgaL._SL250_.jpg",
  "pk": 2,
  "title": "Balls to the Wall",
  "tracks": [
    {
      "albumPk": 2,
      "artistPk": 0,
      "bytes": 5510424,
      "length": "5:02",
      "pk": 2,
      "songName": "Balls to the Wall",
      "unitPrice": 0.99
    },
    {
      "albumPk": 2,
      "artistPk": 0,
      "bytes": 0,
      "length": "3:57",
      "pk": 5090,
      "songName": "Fight it back",
      "unitPrice": 0
    },
    ...
  ],
  "year": 1983
},
{
  "amazonUrl": "http://www.amazon.com/gp/product/B00138KM1U/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=B00138KM1U&linkCode=as2&tag=westwindtec20&linkId=AQAYEWNVF5Z36AZB",
  "artistPk": 2,
  "descript": "An all time classic. At the time Fast as a Shark was THE heaviest thing made to that date...",
  "imageUrl": "https://images-na.ssl-images-amazon.com/images/I/51alnWIHfgL._SL250_.jpg",
  "pk": 3,

```

```

"title": "Restless and Wild",
"tracks": [
  {
    "albumPk": 3,
    "artistPk": 0,
    "bytes": 3990994,
    "length": "3:10",
    "pk": 3,
    "songName": "Fast As a Shark",
    "unitPrice": 0.99
  },
  ...
],
"year": 1982
}
]
}

```

So this object 'composed' by the business object using Foxpro code. How does that work? Lets take a look.

The first bit is the `Load()` method which loads up the Artist entity into an `.oData` member. The code in the core business object selects the record and uses `SCATTER NAME` to push the data into the `.oData` entity object:

```

FUNCTION Load(lvPk)

IF !DODEFAULT(lvPk)
  RETURN .F.
ENDIF

ADDPROPERTY(this.oData,"Albums",NULL)

RETURN .T.

```

The code then dynamically adds an `Albums` property which initially is `null`. The property can be lazy loaded via a `LoadAlbums` method in the `cAlbums` class that returns a collection of albums and tracks for a given artist id:


```

FUNCTION LoadAlbums(lnArtistPk)
LOCAL loAlbum, loBusAlbum, loAlbums

loBusAlbum = CREATEOBJECT("cAlbum")
loAlbums = CREATEOBJECT("Collection")

*** Load Pks then load each item for detailed list
loBusAlbum.GetAlbumPkList(lnArtistPk)

SCAN
  *** Load the top level album entity only
  loBusAlbum.Load(TAlbums.pk)

  *** compose the Track list for each album
  ADDPROPERTY(loBusAlbums.oData,"Tracks",NULL)
  loBusAlbums.oData.Tracks = loBusAlbums.LoadSongs(this.oData.Pk)

  loAlbums.Add( loBusAlbum.oData )
ENDSCAN

RETURN loAlbums

```

These two methods compose a complex nested object structure as an object graph using FoxPro objects and collections to indicate nesting and relationships. Because everything is an object, you can easily add one object or collection to another, dynamically creating any shape you need to represent to the client.

This is a powerful feature that goes a little against FoxPro's cursor centric mindset, but it allows you to express data more naturally than as flat table structures - which if you really want them can still be returned as well. Nothing is stopping you from returning separate artist collections for example (as the .NET server we used earlier does).

Update Request: Updating an Artist

So now lets look updating an article from the client. In update scenarios against a Web Connection server with the wwRestProcess, the endpoint can receive **a single object or value** that is translated into a parameter to the endpoint method.


i Single Parameter Not Enough? Use Object Composition



While a single parameter may sound limiting, remember that you can compose JSON to represent multiple top level objects or values. For example if you wanted to pass 2 parameters you could do pass an object like this:

```
{ parm1: "value 1", parm2: { parm1Value: "value 1.1" } }
```

where parm1, parm2 etc. are your top level 'parameters'. Or you can use an array to represent multiple disparate parameters. **Composition is the key!**


Here's what the request should look like:

 **Request Headers**

 Copy  Run

POST /artist.ms HTTP/1.1

Accept-Encoding: gzip,deflate
Accept: application/json
Content-Type: application/json

 **Request Content**

☒ Formatted json

```
{
  "pk": 2,
  "artistName": "Accept",
  "descript": "With their brutal, simple riffs and aggressive, fast tempos, Accept
    were one of the top metal bands of the early '80s, and a major...",
  "imageUrl": "http://cps-static.rovicorp.com/3/JPG_400/MI0001/389/MI0001389322.jpg
    ?partner=allrovi.com",
  "amazonUrl": "http://www.amazon.com/Accept/e/B000APZ8S4/?_encoding=UTF8&camp
    =1789&creative=390957&linkCode=ur2&qid=1412245037&sr=8-3&tag=westwindtechn
    -20&linkId=KM4RZR3ECUXWBJ6E"
}
```

Here's what the code for the endpoint method looks like.

```

FUNCTION UpdateArtist(loArtist)

IF VARTYPE(loArtist) # "O"
  ERROR "Invalid data passed."
ENDIF

lnPk = loArtist.pk

loBusArtist = CREATEOBJECT("cArtist")
IF lnPk = 0
  loBusArtist.New()
ELSE
  IF !loBusArtist.Load(lnPk)
    ERROR "Invalid Artist Id."
  ENDIF
ENDIF

*** Update just the main properties
loArt = loBusArtist.oData
loArt.Descript = loArtist.Descript
loArt.ArtistName = loArtist.ArtistName
loArt.ImageUrl = loArtist.ImageUrl
loArt.AmazonUrl = loArtist.AmazonUrl

*** Items are not updated in this sample
*** Have to manually update each item or delete/add

IF !loBusArtist.Validate() OR ! loBusArtist.Save()
  ERROR loBusArtist.cErrorMsg
ENDIF

loBusArtist.LoadAlbums()

Serializer.PropertyNameOverrides = "artistName,imageUrl,amazonUrl,albumCount,albumPk,
artistPk,songName,unitPrice,"

RETURN loArt
ENDFUNC

```

This method receives an `loArtist` parameter which is the deserialized JSON from the Artist passed in the request. The code checks and ensures that an object was passed and if so tries to load up the business object's `.oData` from disk. If the Pk is 0 it's a new record, otherwise it's an existing one. If the artist can't be found we throw an `ERROR` which automatically triggers a JSON error response with the error message passed through to the client.

Once we have an Artist instance the object can be updated from the incoming Artist Update data. Once

updated the data is validated and saved using the Business object's internal behavior.

Finally if all went well the object is filled out with albums, to send back to the client. The client can then use the returned object to update its state of the object on its own.

HTTP Verb Overloads

If you've been playing along with this sample, you may have noticed that I used a bit of hand waving in my last example. Can you spot the problem?

The problem is that I used the **same endpoint** for both the `GET` and `POST` operations and also the not yet discussed `DELETE` operation. All of these point at:

```
http://localhost:5200/Artist.ms
```

Web Connection routes requests based on the `.ms` extension and routes to the `Artist()` method, but how do we get to `UpdateArtist()` ? The answer lies in a little bit of logic applied in the `Artist()` method itself that sub-routes requests to the appropriate handlers.

If you recall the `Artist()` method I showed earlier was used for the `GET` operation that returns a single artist. The code left out this extra routing, but I'm going to add it back in now:

```

FUNCTION Artist(loArtist)
LOCAL InId, lcVerb, loArtistBus

InId = VAL(Request.QueryString("id"))
lcVerb = Request.GetHttpVerb()

if (lcVerb == "POST" or lcVerb == "PUT")
    RETURN this.UpdateArtist(loArtist)
ENDIF

IF lcVerb = "DELETE"
    loArtistBus = CREATEOBJECT("cArtist")
    RETURN loArtistBus.Delete(InId) && .T. or .F.
ENDIF

*** GET Operation code below
IF InId == 0
    RETURN this.ErrorResponse("Invalid Artist Id","404 Not Found")
ENDIF

loArtistBus = CREATEOBJECT("cArtist")
IF !loArtistBus.Load(InId)
    RETURN this.ErrorResponse("Artist not found. ","404 Not Found")
ENDIF

*** Lazy load the albums
loArtistBus.LoadAlbums()

Serializer.PropertyNameOverrides = "artistName,imageUrl,amazonUrl,albumCount,albumPk,
artistPk,songName,unitPrice,"

return loArtistBus.oData

```

So if the request is a **GET** request, the code on the bottom runs which retrieves and returns an Artist instance. On **POST** or **PUT** the **UpdateArtist()** method I showed in the last section is called.

Finally there's also inline logic for deleting an Artist using the **DELETE** verb.

HTTP Verb overloading is a common concept in REST - you use nouns (ie. Artist) in the URL to describe a thing or operation, and a verb (**GET** , **POST**) to describe what to do to it. The combination of the two - **URL + HTTP Verb - make up the unique endpoint.**

Server Error Handling

No discussion of services is complete without giving some thoughts to error handling. Incidentally this is one of my pet peeves because there are plenty of services out there that do a horrible job of error handling, not

providing decent error information back to the client.

It's important that your application provides meaningful error information. There are a number of things that can be returned to let the client know what's going on:

- HTTP Status Codes
 - 200's success
 - 300's forwarding
 - 400's Authorization, Not Found, Invalid etc.
 - 500's errors
- Error Response JSON

Status codes should be the first line of response. If you have an authorization request that fails, a `401 Not Authorized` response is appropriate. If a resource is not found a `404 Not Found` should be returned. On the client these show as errors but offer a quick way to know what went wrong. If your application crashes while processing it should return a `500 Server Error` response.

If you can it's also useful to return an error response from a request that **provides error information in a consistent manner**. I like to return a consistent error structure that includes an `isError` property, and a `message` property at minimum and then add additional fields as needed. For example, in debug I might want to send a stack trace so I can tell where the code failed.

Web Connection REST automatically handles a number error scenarios automatically in the `wwRestProcess` class. If you access an invalid URL it automatically returns a `404 Not Found` error. If automatic authentication in Web Connection fails it automatically sends a `401 Unauthorized`. And any hard failures in your method code that aren't trapped return a `500 Server Error` result. All of these failures also return a JSON error object.

To check this out lets break some code. For this to work we have to make sure Web Connection's `Server.IDebugMode = .F.` (or via UI configuration) or else any error will break into the source code.

If I change the `Artists()` method to include an invalid method call which is a 'generic application error' like this:

```
FUNCTION Artists()  
  
  loArtistBus = CREATEOBJECT("cArtist")  
  lnArtistCount = loArtistBus.GetAArtistList()  
  
  Serializer.PropertyNameOverrides = "artistName,imageUrl,amazonUrl,albumCount,"  
  
  RETURN "cursor:TArtists"  
ENDFUNC
```

I get the following error response:

Internal Server Error

Request Headers

CopyRun

GET http://localhost:5200/Artists.ms HTTP/1.1

Accept-Encoding: gzip,deflate
Accept: application/json
User-Agent: West Wind HTTP .NET Client

Response Headers

76 bytesTime: 144msFirst byte: 137ms

HTTP/1.1 500 Internal Server Error

Date: Tue, 28 Sep 2021 00:02:27 GMT
Server: Kestrel
Transfer-Encoding: chunked
RequestId: 83_4defea5e
Content-Type: application/json; charset=utf-8

Response Content

```
{  
  "isCallbackError": true,  
  "message": "Property GETAARTISTLIST is not found."  
}
```

Notice the response is `500 Server Error` and I get the FoxPro error message in the JSON response. Any error that occurs in your method basically triggers a 500 response. This is a good way to ensure unhandled errors provide some feedback rather than just a server error page.

You can also explicitly force an error response. There are two ways to do this:

- Simple call `ERROR "<lcErrorMessage>"`
- `THIS.ErrorResponse()`

The `ERROR` call is handled the same way as an unhandled exception except it uses your error message. The `ErrorResponse()` method allows you to specify a message as well as a status code. So rather than always returning a generic `500` response you can be more specific about the result code. Many errors are of the `404 Not Found` kind or `204 No Content`. You can [find HTTP status codes here](#).

With that in mind let's pretend we have a problem loading the Artist list and return an error to the client:

```

FUNCTION Artists()

loArtistBus = CREATEOBJECT("cArtist")
InArtistCount = loArtistBus.GetArtistList()

IF .T. && InArtistCount < 0
    * ERROR "Couldn't retrieve artists"  && 500 error
    THIS.ErrorResponse("Couldn't retrieve artists","404 Not Found")
    RETURN
ENDIF

Serializer.PropertyNameOverrides = "artistName,imageUrl,amazonUrl,albumCount,"

RETURN "cursor:TArtists"
ENDFUNC

```

CORS for Web Browser Access

If you're building Web Server APIs that are going to be accessed directly by a Web Browser, you need to set up CORS (Cross Origin Resource Sharing) on the server. CORS is a URL restriction protocol that the server provides and the client checks to see whether the server allows the client's origin domain to connect and retrieve data from the server. This is a crazy double blind backwards protocol, that is enforced solely by browsers and totally useless for any other HTTP client. However, browsers **require a server CORS host policy** in order to allow connecting to a non-local Web site for `fetch` or `XHR` requests.

In Web Connection CORS can be enabled with the following code in a `wwProcess` class' `InitProcess()` method:


```

FUNCTION OnProcessInit
LOCAL lcOrigin, lcVerb

*** Explicitly specify that pages should encode to UTF-8
*** Assume all form and query request data is UTF-8
Response.Encoding = "UTF8"
Request.IUtf8Encoding = .T.

lcOrigin = Request.ServerVariables("HTTP_ORIGIN")
IF !EMPTY(lcOrigin)
!* *** Add CORS header to allow cross-site access from other domains/mobile devices on Ajax calls
!* Response.AppendHeader("Access-Control-Allow-Origin","*") && all domains always
Response.AppendHeader("Access-Control-Allow-Origin",lcOrigin) && requested domain - effectively
all
Response.AppendHeader("Access-Control-Allow-Methods","POST, GET, DELETE, PUT,
OPTIONS")
Response.AppendHeader("Access-Control-Allow-Headers","Content-Type, *")
*** Allow cookies and auth headers
Response.AppendHeader("Access-Control-Allow-Credentials","true")
ENDIF

*** CORS headers are requested with OPTION by XHR clients. OPTIONS returns no content
lcVerb = Request.GetHttpVerb()
IF (lcVerb == "OPTIONS")
*** Just exit with CORS headers set
*** Required to make CORS work from Mobile devices
RETURN .F.
ENDIF

RETURN .T.

```

This explicitly allows access to all 'origins' which are essentially domains. The 'origin' is a base URL from a source site that is typically sent by a client browser when making `fetch()` or `XmlHttpRequest` HTTP calls to a non-native domain. So if I'm running on `west-wind.com` and I want to call an API on `foxcentral.net` from a browser, CORS policy has to explicitly allow `https://west-wind.com` access. You can specify either a comma delimited list of domain, or a wildcard that allows all domains using `*` as I'm doing above.

Note that this policy is a browser security feature and only applies to Web browser calls to non-local domains. It has no effect on a FoxPro HTTP client for example, but the server has to send these headers regardless to ensure that Web clients can consume the data.

Summary

Alright, in this article I've shown you both how to call JSON REST services from FoxPro and how to create JSON REST services with FoxPro code. API Services are very powerful and give you a lot of options for publishing data in a fairly easy to create fashion. JSON as a message format is a great tool as it is relatively easy

to create and parse in FoxPro. It has none of the complications that XML and SOAP suffered from - there's no ambiguity about the simple types that JSON provides.

HTTP tools on the client are available on just about any platforms - often with many options. On Windows you can use raw WinHttp() calls, use .NET for passthrough HTTP calls, or if you want more control a full featured library like `wwHttp` can offer a number of nice helper features to make it easy to send and receive content between client and server.

You can also create REST services fairly easily using any of the existing Web server solutions that you might already be using. Because JSON is a fairly simple format to create and parse, any existing solution can provide REST functionality with a little manual work, or you can use a ready made framework like the `wwRestProcess` class in Web Connection that abstracts the entire process for you and turns REST endpoints into simple methods with an input parameter and result value.

REST is no longer new technology, but it's had staying power and there doesn't appear to be anything set to replace it in the foreseeable future. Part of this is because the simplicity of the tech just works and easy to implement. There are many patterns like Micro Services, Serverless Computing, and countless Cloud Services that are all just slight variations of the REST service technology. These approaches are here to stay and building on them both provides benefits in usage, as well as

Resources

Links

- [Samples and Documents](#) (GitHub)
- [AlbumViewer Web Site](#) (.NET Core site and API)

Tools and Libraries

- [West Wind Web Connection](#)
- [West Wind Client Tools](#)
- [West Wind Web Surge](#)
- [Postman](#)
- [JSONView Chromium Addin](#)



this article created and published with the [Markdown Monster Editor](#)