A MiSite can be configured to provide RESTful web services. This document describes the steps necessary to do so. While we will cover some RESTful concepts, this is not tutorial on RESTful web services themselves – for a good overview of RESTful web services, please see http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069.

Web Service Technologies

A web service is a method of communication between two processes over a network. While a web service does not have a GUI interface, GUI interfaces that interact with web services through their APIs are common. For example, a web page that prompts for a postal code may use a web service for validation, but it's the web page that provides the GUI, not the web service.

Currently, there are two predominant web service technologies – SOAP-based and REST. SOAP-based web services use XML-based protocols for message exchange. Dyalog provides support for SOAP-based web services with SAWS, the Stand Alone Web Services framework. REST (Representational State Transfer) web services have become popular for providing public APIs. REST is an architectural style, unlike SOAP which is a standardized protocol. REST makes use of HTTP, and does not create any new standards. It can structure data into XML, YAML, or any other machine readable format, but usually JSON – JavaScript Object Notation – is preferred. REST is very data-driven, compared to SOAP, which is strongly function-driven. A web service that uses REST is termed "RESTful".

The 5 Minute Oversimplified Guide to REST

This section is intended to give you a very basic understanding of REST. REST implements a verb-noun style of interaction.

REST has the following characteristics:

- Client-Server: a pull-based interaction style: consuming components pull representations.
- Stateless: each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- Cache: to improve network efficiency responses must be capable of being labeled as cacheable or non-cacheable.
- Uniform interface: all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, DELETE).
- Named resources - the system is comprised of resources which are named using a URL.
- Interconnected resource representations - the representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another.
- Layered components - intermediaries, such as proxy servers, cache servers, gateways, etc, can be inserted between clients and resources to support performance, security, etc.

Verbs

The verbs in REST are the set of HTTP commands. If you're familiar with web site technology, you probably already know about the HTTP commands **GET** and **POST**. The HTTP protocol has several other commands that are often used for RESTful web services. The first four verbs are typically used to

implement CRUD – the operations CREATE, READ, UPDATE, and DELETE.

Verb (HTTP Command)	Common RESTful Use
GET	READ – retrieve a resource
POST	UPDATE – update an existing resource
PUT	CREATE – create a new resource
DELETE	DELETE – delete a resource
HEAD	check if a resource exists
OPTIONS	query what verbs are available for a resource

Nouns

The nouns in REST are resources and are accessed via URIs (Uniform Resource Identifiers).

A resource is some collection of data. It could be a list of customers, information about a specific customer, a list of orders for a specific customer, the order details for a specific order for a specific customer,... you get the idea.

A URI is simply a string of characters to identify a resource. You're probably most familiar with URLs, which is a type of URI that identifies a web address - typically a file somewhere on the web. A URI doesn't necessarily have to point to an actual physical resource like a file, it merely identifies an entity that is considered a resource. For instance, the URI http://someWebService.com/customer/123 might identify the customer with id 123 though there may not be an actual physical file for customer 123.

A Few Observations

- REST is very flexible and you have the power to implement (or not) whatever set of verbs upon whatever resources make sense for your application. For instance:
  POST http://myservice/Persons/ might add a new record to the Persons table in your application, but
  POST http://myservice/ may not make any sense in the context of your application.
- Many web services that claim to be "RESTful" have aspects that violate REST principles. In fact some RESTful advocates condemn such practices with almost religious fervor.

MiServer gives you a framework to flexible design and develop web services that use standard HTTP commands and can return results in a variety of formats. How "purely" RESTful you care to make your web service is entirely up to you.

Implementing a RESTful Web Service in MiServer

Configuration

In your MiSite configuration folder, create or update the following entries in Server.xml.

- Set RESTful to 1 – this is important so that MiServer knows how to interpret URIs.
  For example, the URI http://myservice/Persons/Brian
  RESTful≠1 – look for the file /Persons/Brian
  RESTful=1 – look for the file /Persons and supply "Brian" as a parameter

- Set AllowedHttpCommands to a comma-delimited string of the verbs you wish to support in your web service.

```
<Server>
   <RESTful>1</RESTful>
   <AllowedHTTPCommands>get,put,delete,post</AllowedHTTPCommands>
</Server>
```

It is strongly recommended that you update /Config/Server.xml file in your MiSite folder and not at the MiServer level.

RESTful Page Requirements

Use the `RESTfulPage` *base class for your pages*

Instead of using **`MiPage`**, or some other base class, you will use **`RESTfulPage`** as the base class for your RESTful pages.

```
:Class myService : RESTfulPage
:EndClass
```

Use `Respond` *instead of* `Compose`

MiServer expects a **`RESTfulPage`** page to have a public method named **`Respond`** which returns the results for your web service.
`Respond` must be niladic and return a result.

```
∇ r←Respond
  :Access public
  ⍝ your code here
∇
```

Features of the RESTfulPage Base Class

The **RESTfulPage** base class shares many features of the **MiPage** class.

- data elements passed with the HTTP request, either via query parameters in the URI, or in the message body are stored in **_PageData** and are accessible with the **GET** method.
- **_Command** returns the lowercased HTTP command
- **_URI** returns a vector of vectors containing the segments of the URI
  e.g. for the URI /customer/123/order, **_URI** will contain

  _URI

customer	123	Order

  Note: the first element is the name of your RESTful page.
- **The SetStatus** method is used to set the HTTP status code (for more detail about HTTP status codes see: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html).
  Note: Status 200 (OK) is set by default; you only need to set the status when something goes wrong, like a resource not being found (status 404).
- **The SetHeader method is used to set HTTP headers in the response. MiServer takes care of setting most headers like content-length and content-type for you.**

Examples

Simple Read Only Loan Calculation Service

The page below implements a loan calculation web service.
The parameters are provided in the query string of the URI.
The results are stored in a namespace, which MiServer converts into a JSON object in the response.

```
[0]    :Class mortgagews : RESTfulPage
[1]    ⍝ payment calculation
[2]        calcpmt←{0::'Error' ⋄ p r n←⍵÷1 1200 (÷12) ⋄ .01×⌈100×p×r÷1-(1+r)*-n}
[3]    ⍝ principal calculation
[4]        calcprin←{0::'Error' ⋄ r n m←⍵÷1200 (÷12) 1 ⋄ .01×⌈100×m÷r÷1-(1+r)*-n}
[5]
[6]        ∇ response←Respond;mask;r ⍝ render the initial page
[7]          :Access Public
[8]          r←⎕NS ''
[9]          r.msg←'Please provide either (prin rate term) or (rate term pmt)'
[10]         :If 1=+/mask←0∘≡¨r.(prin rate term pmt)←⍬ Get 'prin rate term pmt'
[11]             :If mask[1] ⍝ principal missing
[12]                 r.(msg prin)←'' (calcprin r.(rate term pmt))
[13]             :ElseIf mask[4] ⍝ payment missing
[14]                 r.(msg pmt)←'' (calcpmt r.(prin rate term))
[15]             :EndIf
[16]         :EndIf
[17]       ∇
[18]   :EndClass
```

Sample Request: GET localhost:8080/mortgagews?prin=100000&rate=.05&term=30

Response: `{msg:"",pmt:279.88,prin:100000,rate:0.05,term:30}`

Discussion:
Line [0] declares the mortgagews class which is based on the **RESTfulPage** base class.
Lines [2] and [4] define functions for payment and principal calcucations
Line [6] begins the definition of the mandatory Respond method
Line [7] Don't forget the method needs to be public!
Line [8] Initializes the namespace that will contain the result(s) of the web service
Line [9] Sets a default message (the msg element in the namespace)
Line [10] transfers the elements passed in the URI to the result namespace and checks that only one element in the calculation is missing
Lines [11] and [13] check if the missing element is either the principal or the payment
Lines [12] and [14] perform the calculation for the missing element and set the result msg and pmt elements

MiServer converts the namespace returned by **Respond** into a JSON structure which it sends in the response to the client.

A More Comprehensive Example

The **RESTful** sample MiSite found at https://github.com/Dyalog/MiSites/RESTful contains the **mortgagews** web service presented earlier as well as more complex web service, **customer**, that implements a small customer and order database.
The zip file https://github.com/Dyalog/MiSites/blob/master/RESTful.zip contains a zipped copy of the MiSite and can be downloaded by:

1) Navigating to the link for the zip file
2) Clicking the "Raw" button in GitHub
3) Saving the zip file to your local drive

Once you have downloaded and unzipped the file, you may start the web service by loading the **miserver** workspace and entering:

```
Start 'path/RESTful'
```

Where *path* is the destination path where you unzipped RESTful.

The file **demorest.txt** in the *path*/**RESTful** folder contains a demonstration that can be executed using the **]demo** user command.

```
]demo path/RESTful/demorest.txt
```

The demonstration will walk through examining various aspects of a MiServer-based RESTful application. Some of those aspects are presented in the material that follows.

The RESTful MiSite represents one approach to implement a RESTful web service; you have the flexibility to implement your web service using this or any other approach you find more appropriate.

Config/Server.xml

The **Server.xml** file contains settings for the RESTful MiSite.

```
<Server>
    <Name>MiServer Sample RESTful Application</Name>
    <ClassName>SimpleSampleServer</ClassName>
    <RESTful>1</RESTful>
    <AllowedHTTPCommands>get,put,delete,post,options</AllowedHTTPCommands>
</Server>
```

<Classname> defines the name of the class (**SimpleSampleServer**) that is based on the **MiServer** base class. Classes derived from **MiServer** allow the user to customize server behavior.

SimpleSampleServer

```
:Class SimpleSampleServer : MiServer
    ∇ make args
      :Access Public
      :Implements Constructor :Base args
    ∇

    ∇ onServerStart
      :Access public override
      ⍝ Create the sample "database"
      #.bl.Init
    ∇
:EndClass
```

SimpleSampleServer overrides the **onServerStart** method to run **#.bl.Init** which initializes the "database" for the web service. **#.bl** is a namespace which implements the business logic for our web service and is located in the **path/RESTful/Code** folder. Any APL script files located in this folder are loaded during MiServer initialization.

HTTPCmd

Also located in the **path/RESTful/Code** folder is the **HTTPCmd** utility from Conga (Dyalog's TCP/IP toolkit). **HTTPCmd** is necessary to test the web service from APL. **HTTPCmd** is an operator that can issue any HTTP command.

As you step through the demostration, you will see the following examples:

```
('options' HTTPCmd) 'localhost:8080/customer/'  ⍝ retrieve the documentation
('get' HTTPCmd) 'localhost:8080/customer/'      ⍝ retrieve the customer list
('get' HTTPCmd) 'localhost:8080/customer/1200'  ⍝ retrieve customer 1200
('get' HTTPCmd) 'localhost:8080/customer/1200/order/' ⍝ all orders for customer 1200
('get' HTTPCmd) 'localhost:8080/customer/1200/order/9/' ⍝ order details for order 9
('get' HTTPCmd) 'localhost:8080/customer/0'   ⍝ non-existent customer
('post' HTTPCmd) 'localhost:8080/customer/' ('name' 'Nick') ⍝ add a new customer
('put' HTTPCmd) 'localhost:8080/customer/400/' ('name' 'Dan') ⍝ update a customer
('delete' HTTPCmd) 'localhost:8080/customer/1200/'  ⍝ delete customer 200
```

Customer.Respond

The **Respond** method in the **customer** MiPage examines the HTTP command for the request and calls the appropriate method for the operation.

```
∇ r←Respond;custid;urlParams
  :Access public
  r←''
  :Select _Command  ⍝ which HTTP command?
  :Case 'post' ⋄ r←AddCustomer       ⍝ create new customer
  :Case 'get' ⋄ r←GetCustomer        ⍝ retrieve
  :Case 'put' ⋄ r←UpdateCustomer     ⍝ update
  :Case 'delete' ⋄ r←DeleteCustomer  ⍝ delete
  :Case 'options' ⋄ r←Documentation  ⍝ documentation
  :Else ⍝ invalid command
      SetStatus 400 ⍝ Bad Request
      →0
  :EndSelect

  :If 0∊⍴r
      SetStatus 404 ⍝ Not Found
  :EndIf
∇
```

customer.GetCustomer

The getCustomer method does the most analysis of the URI to determine what action to perform.

```
∇ r←GetCustomer
:Select ⊃⍴_URI
  :CaseList 0 1
    :If ~0∊⍴r←#.bl.GetCustomer _URI
      r←'custid' 'custname' 'custURI'Annotate r,makeCustomerURI¨r[;1]
    :EndIf
  :CaseList 2 3   ⍝ GET customer/custid/order or customer/custid/order/orderid
    :If 'order'≡2⊃_URI
      :If ~0∊⍴r←#.bl.GetCustomerOrders 1⊃_URI
        :If 2=⍴_URI ⍝ retrieve list of orders for customer
          r←'orderid' 'custid' 'orderdate'Annotate r
        :ElseIf ~0∊⍴r←(1⊃_URI)#.bl.getCustomerOrder(3⊃_URI)
          r←'order' 'details'#.JSON.toNS('orderid' 'custid' 'orderdate')('qty'
                                         'productid' 'productname')Annotate¨r
        :EndIf
      :EndIf
    :Else
      SetStatus 400 'Invalid format'
    :EndIf
:EndSelect
∇
```