WEB SERVICE PART 1

By Danny Wind

starter — expert

INTRODUCTION

This series of articles is about writing your own web services server and client in Delphi. The approach of all articles is pragmatic. This first article introduces some of the concepts you need to know and shows you how to create and consume your own web service in Delphi.

What is a web service? A web service sends and receives data over the world wide web. Web services mostly communicate over the internet through the HTTP protocol and send and receive data in one of the web formats, such as JSON, XML or HTML.

Why would you want to build a web service? Web services are used in almost every app, website and desktop application to get local and remote data. Web services are also used for interoperability and import and export. For instance accounting software usually has an interface to its locally or remotely stored data using a local or remote web service. Web services are also easily scalable. Start with a simple locally run web service on a laptop and then scale up into a server park or into the cloud.

In a sense even a website is a web service. When you open an URL (`https://www.blaisepascalmagazine.eu/`) in your web browser the browser sends a HTTP GET command to the website URL and in response receives data in the form of a HTML page.

However, when software developers mention web services, they mostly mean a web service that uses REST (REpresentational State Transfer) over HTTP to send and receive JSON formatted data. For instance the DuckDuckGo API that resolves your question into a direct answer from WikiPedia, GitHub and more. Just try this link in your browser

```
https://api.duckduckgo.com/
?q=Blaise&format=json&pretty=1
```

When you access a web service you supply an URL combined with an identifier into an URI for a specific resource.

URL - Uniform Resource Locator
This is the human readable address that a resource (a web service) can be found at. An example would be https://duckduckgo.com/. It's translated to a physical IP address through DNS. This way the resource can be located over a TCP/IP network. In analogies an URL would be the home address for the house where your resource lives.

URI - Uniform Resource Identifier
There is also a thing called URI. This identifies a specific resource. If you just go with the idea that this adds a specific resource identifier to retrieve from the URL location, you're not far off. An example of an URI is https://duckduckgo.com/index.html. In analogies the URI would be a specific bookcase inside the house.

URL vs URI
It's fun to debate what is or isn't an URL, or an URI, because the definition in the RFC documents leaves some of it open to different interpretations. So you may get into a discussion at the coffee machine on URL vs URI, after which you can relax with your cup of coffee, as it really doesn't matter. It's just the location of your web service.

The protocol used is HTTP, denoted by the prefix http:. Even if the URL has HTTPS in its prefix this is still talking HTTP, it's just encrypted.

HTTP - Hyper Text Transfer Protocol
HTTP is the protocol used to communicate over TCP/IP with your web service. In analogies HTTP is a very limited language used to exchange data.

HTTP has nine commands that are used to request and receive data. Each of these commands has a specific purpose. For web services there are four commands you should start with.

HTTP GET idempotent, cacheable	Retrieves data from the resource
usage in our web service	**SELECT** (get existing record, disallow caching so we get new data each time)
HTTP POST not idempotent, not cacheable/stale	Appends data to the existing resource
usage in our webservice	**UPDATE** existing (partial update of fields in a record, not updating the primary key)
HTTP PUT idempotent, not cacheable/stale	Replace the existing resource or inserts data as a new resource
usage in our web service	**INSERT** new (or **REPLACE**) (insert new record with new primary key, or replace entire record)
HTTP DELETE idempotent, not cacheable/stale	Deletes the resource
usage in our web service	**DELETE** (delete existing record or return error if it doesn't exist)

In our article we will use a simplified mapping of HTTP commands to actions we want our web service to perform. For more complete mapping you could take a look at some of the open source REST web service frameworks available.

Some Open Source web service frameworks
MARS Curiosity Framework:
`https://github.com/andrea-magni/MARS`
mORMot ORM Framework
`https://github.com/synopse/mORMot`
WiRL RESTful library
https://github.com/delphi-blocks/WiRL

We will not be using these existing frameworks, instead we will be building a simple web service server and client from scratch.

STARTING THE BUILDING OF THE PROGRAM on the next page we will create the web service client.

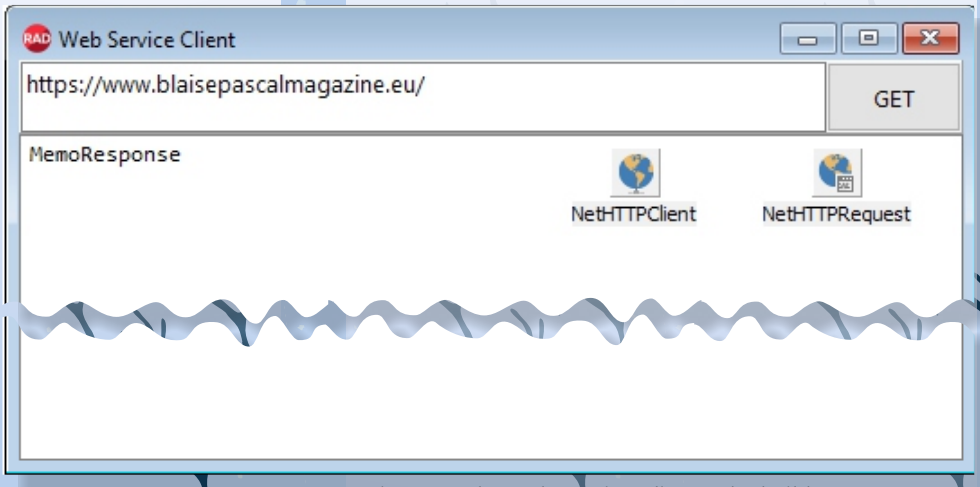


Figure 1: The Web Service Client to be build

Start with either a VCL or FireMonkey application

1. Add a Panel and align Top
2. Add a Button to this Panel and rename to ButtonGet, align Right
3. Add an Edit to this Panel and rename to EditURL, align Client
4. Add a `NetHTTPClient` to the Form
5. Add a `NetHTTPRequest` to the Form and link the Client property to `NetHTTPClient`
6. Add an `OnClick` event-handler to the `ButtonGet`

```
procedure TFormMain.ButtonGETClick(Sender: TObject);
begin
 NetHTTPRequest.MethodString := 'GET';
 NetHTTPRequest.URL := EditURL.Text;
 NetHTTPRequest.Execute();
end;
```

7. The `HTTP GET` request is executed asynchronously, which means the response will appear at some time in the future. When the response arrives the `OnRequestCompleted` event-handler of the `NetHTTPRequest` will be called.
8. Add an `OnRequestCompleted` event-handler to the `NetHTTPRequest`

```
procedure TFormMain.NetHTTPRequestRequestCompleted(const Sender: TObject;
 const AResponse: IHTTPResponse);
begin
 MemoResponse.Text := AResponse.ContentAsString;
end;
```

9. Run and test using a website URL, for instance
   `https://www.blaisepascalmagazine.eu/`
10. The GET will return a HTML page

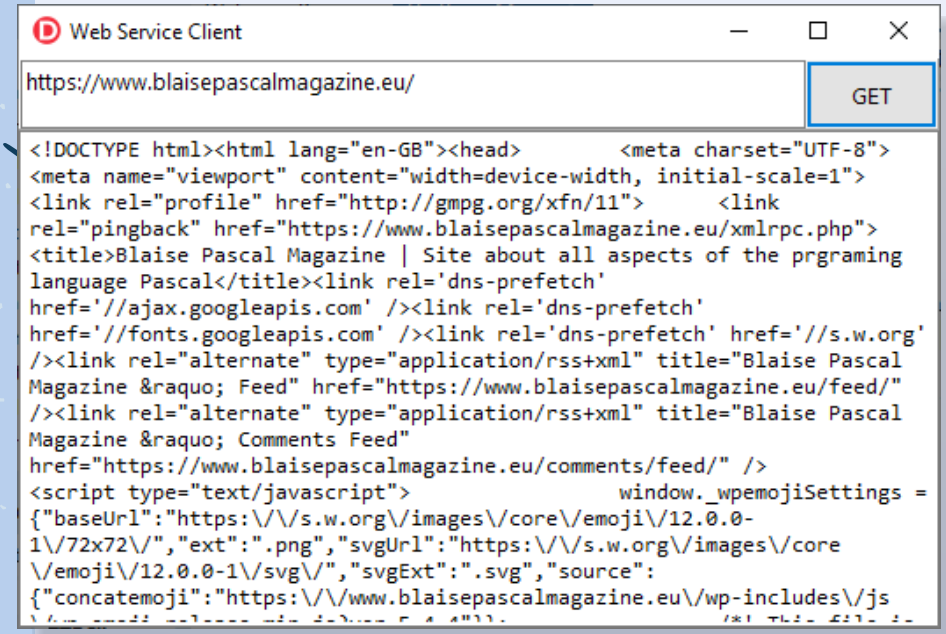


Figure 2: The result of the GET order event

12. When you use an URL to a REST endpoint from an existing Web Service, you get JSON data. Try this URLs in the Web Service Client

    `https://api.discogs.com/artists/457265`

13. The result is JSON data, recognizable due to the curly braces {}

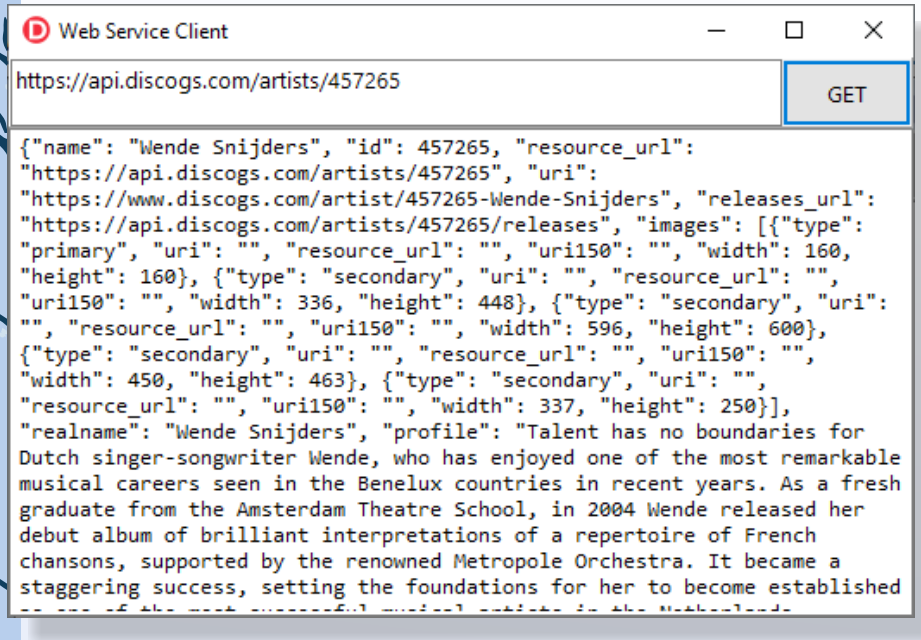


Figure 3: The result is JSON data

14. The next step is to create our own Web Service Server

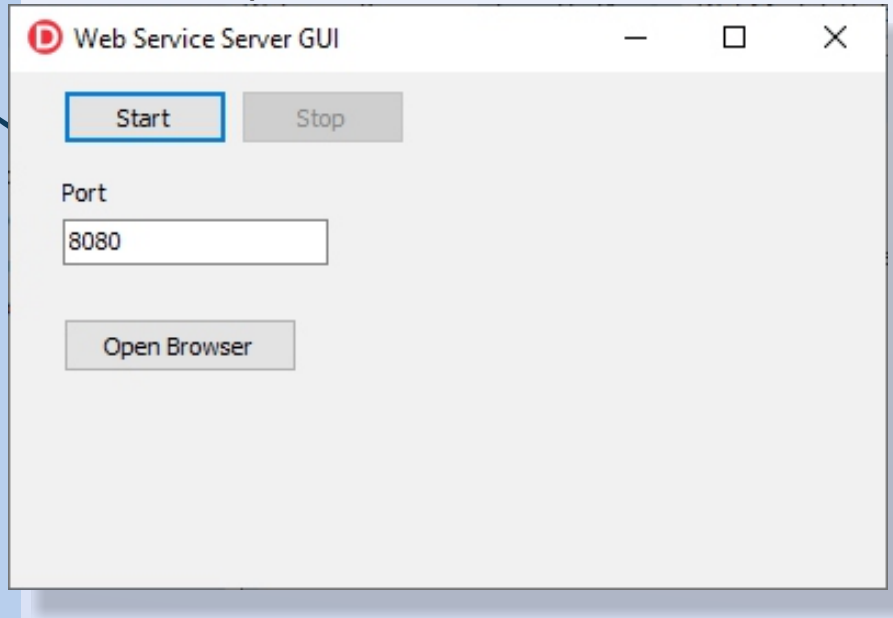


Figure 4: The Web Service Server Gui

15. We start with the New Item wizard for Web | Web Server Application

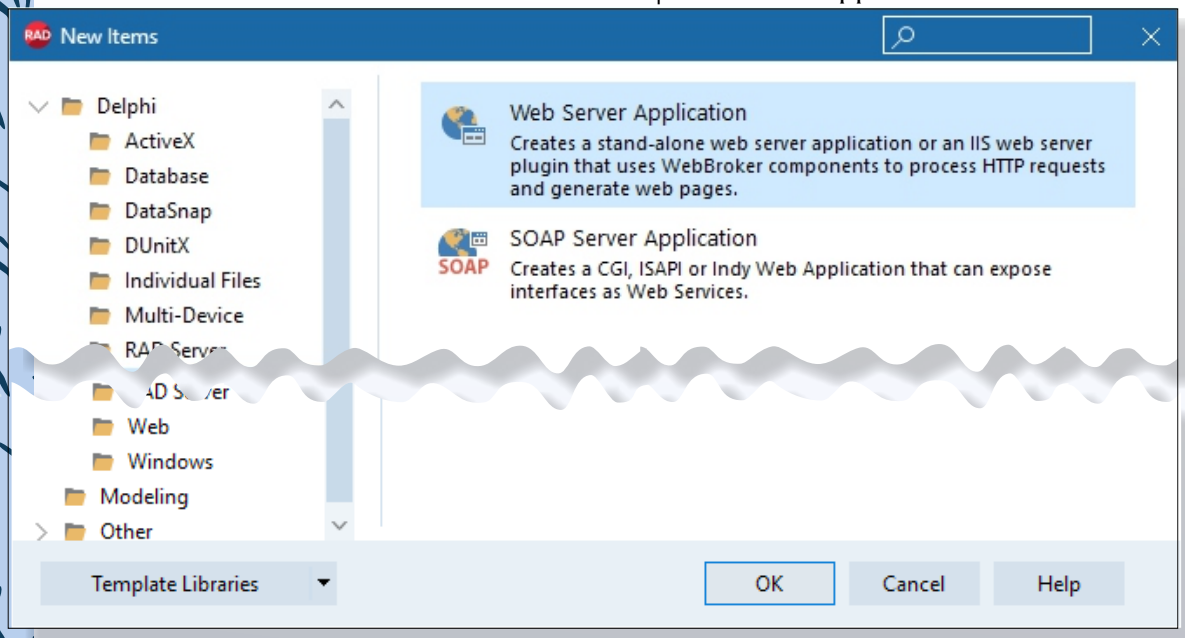


Figure 5: starting with the Web Server Appication: New →Other→Web

16. This will allow you to create a web server that runs standalone or as a library in **Apache (Linux)** or **IIS (Windows)**. It will process HTTP requests and you can write the code on how it should respond, with plain text, a JSON string, HTML page or even an image or a file.

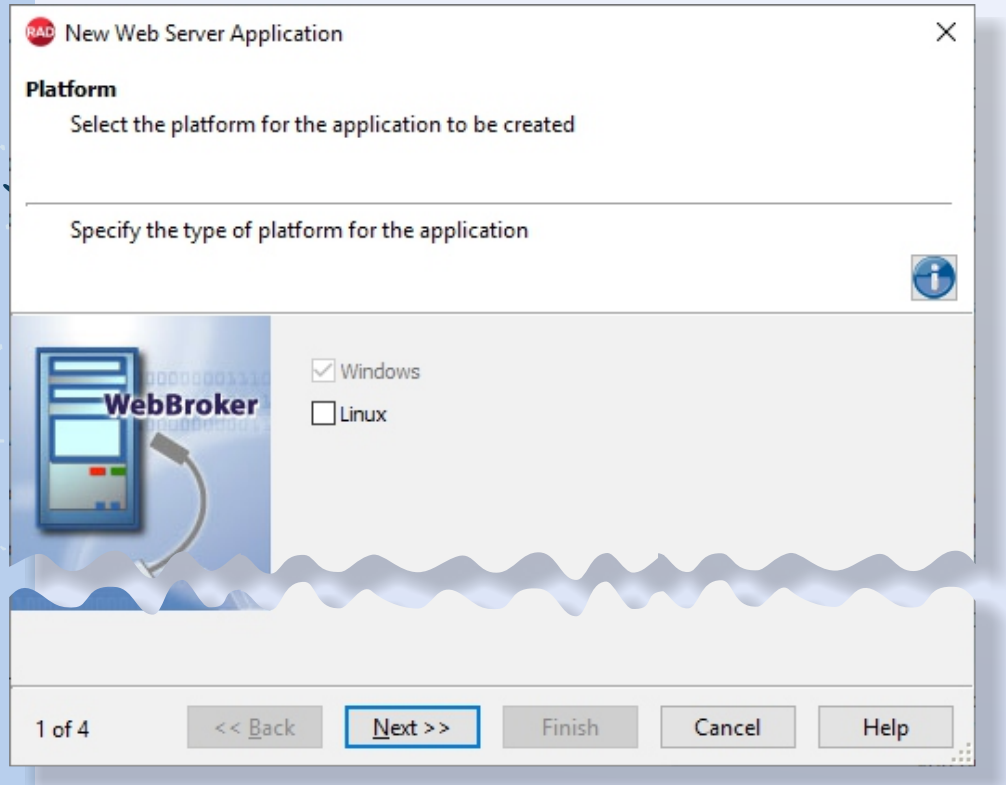


Figure 6: Standard for Windows

17. For now we go with Windows. We can add Linux support later on.

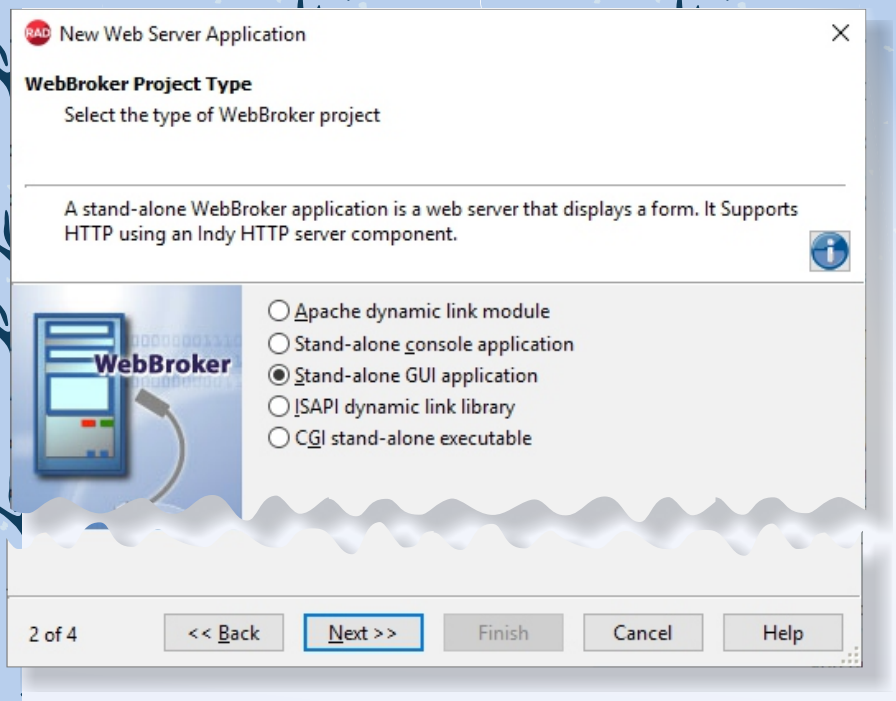


Figure 7: Select the type of WebBroker

18. For now we create a stand-alone GUI application. If we run this wizard again later on we can combine these options into one Project Group and share the base code files between these options.

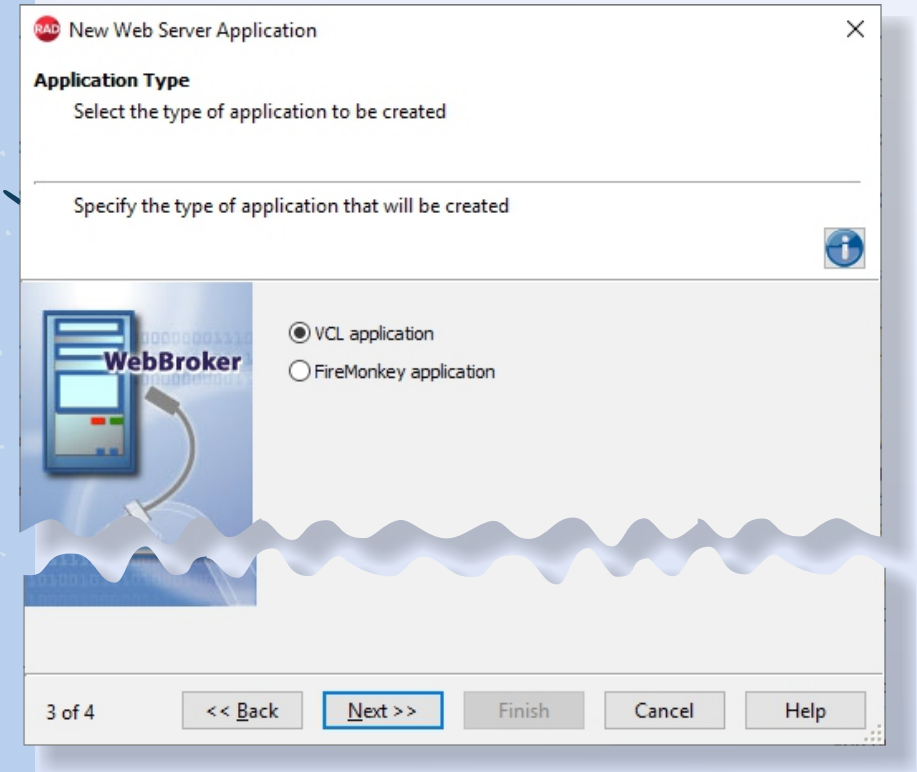


Figure 8: VCL

19. VCL is OK

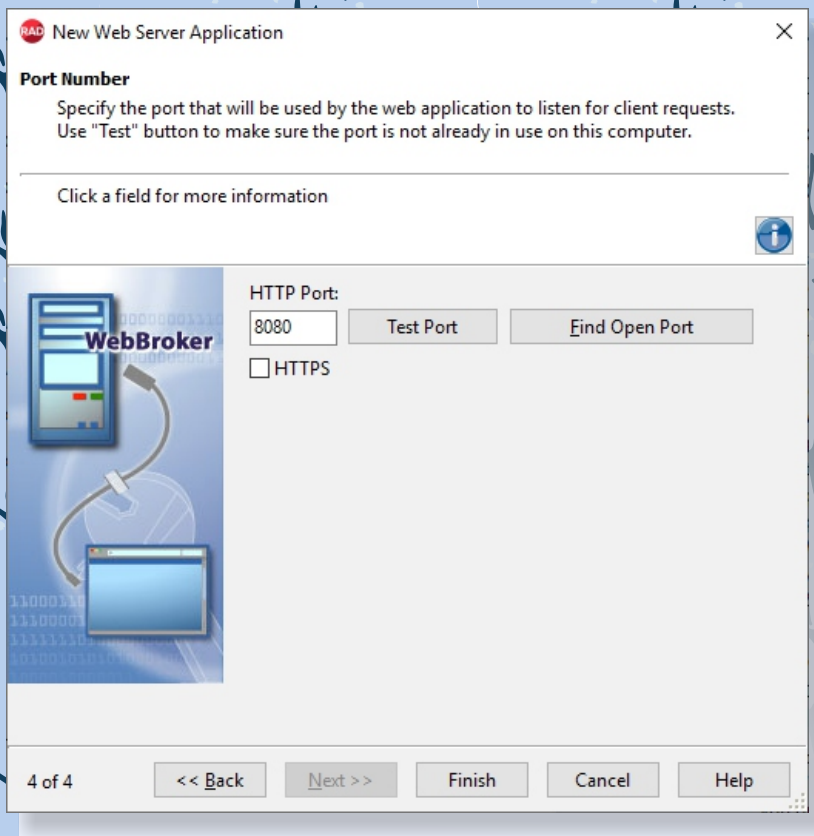


Figure 9: The port number

20. Test if the default port is not already taken on your machine.
    If it is you can Find Open Port or just try another port, such as 8088 or 8888.
21. Save the files into a separate directory and save the Project as WebServiceServerGUI

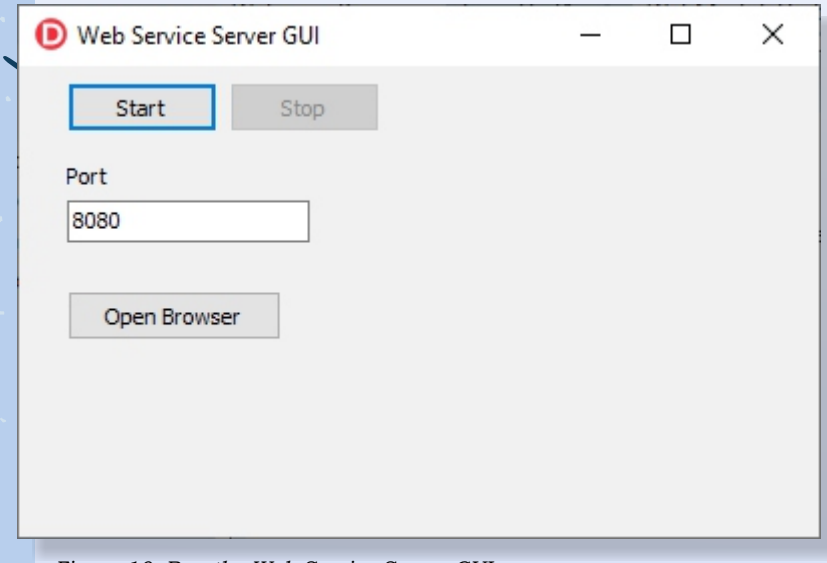


Figure 10: Run the Web Service Server GUI

22. Run the Web Service Server GUI
23. Click on Start, this should open the Windows Firewall configuration, depending on your local network configuration (Private or Public) you can choose to open the port for private networks only or for public networks (if you have your computer configured as such). Please note that when opening it up for public networks would mean the port is also open when you visit an airport.

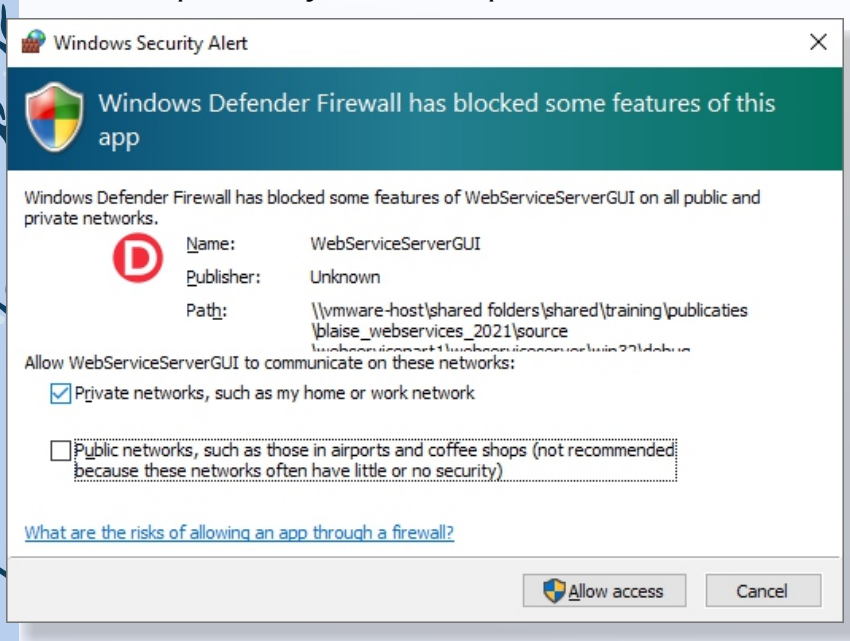


Figure 11: The Windows Firewall config

24. Make note of the URL that is then opened in the web browser
    `http://localhost:8080/`
25. Run the Web Service Client and open this URL.
26. The :8080 in this URL is the internet port. An internet port is like a door, each door has its own number. If the port is opened in the firewall, traffic is allowed through.
27. The localhost in this URL translates to a loopback to this machine. It's translated to a local IP address (127.0.0.1) that points to the machine you are running on.

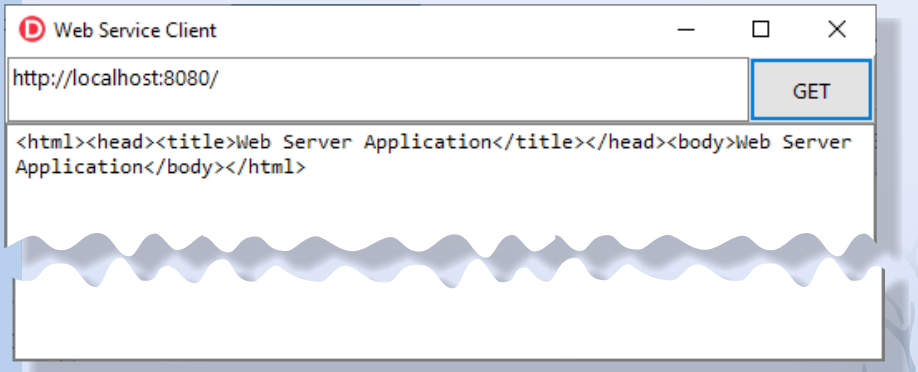


Figure 12: Run the Web Service Client and open this URL.

28. This is the default handler for the Web Server Service
29. Let's add a new GET handler to the Web Server Service
30. Open the Web Server Service GUI project and open the WebModuleUnit

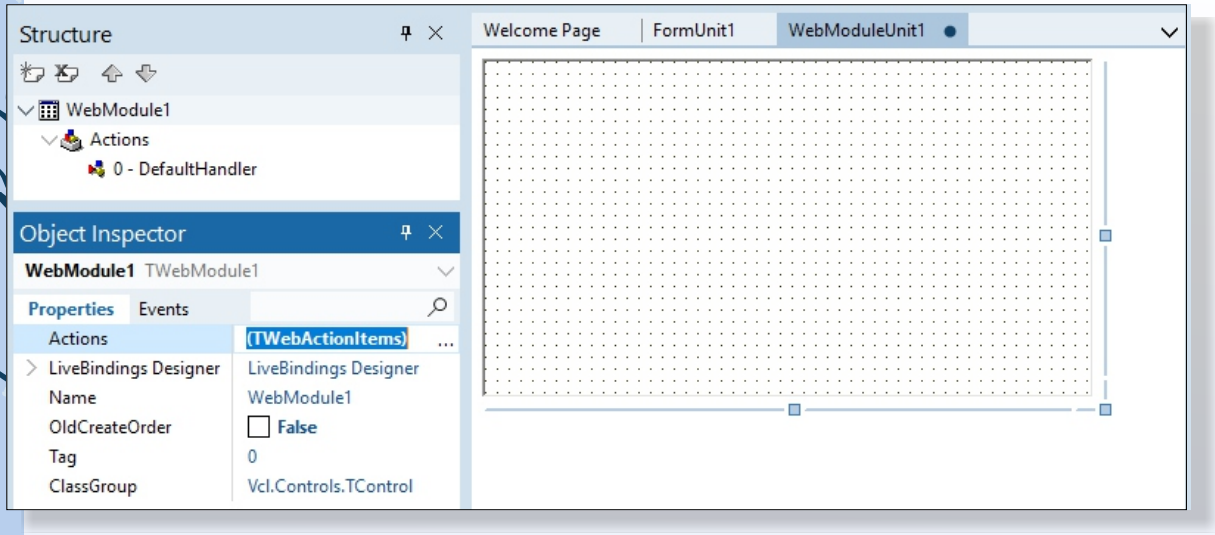


Figure 13: WebModuleUnit

31. HTTP requests to the Web Server are mapped to WebActionItems (Action Handlers). Add a new WebActionItem by clicking the three-dots in the property Actions

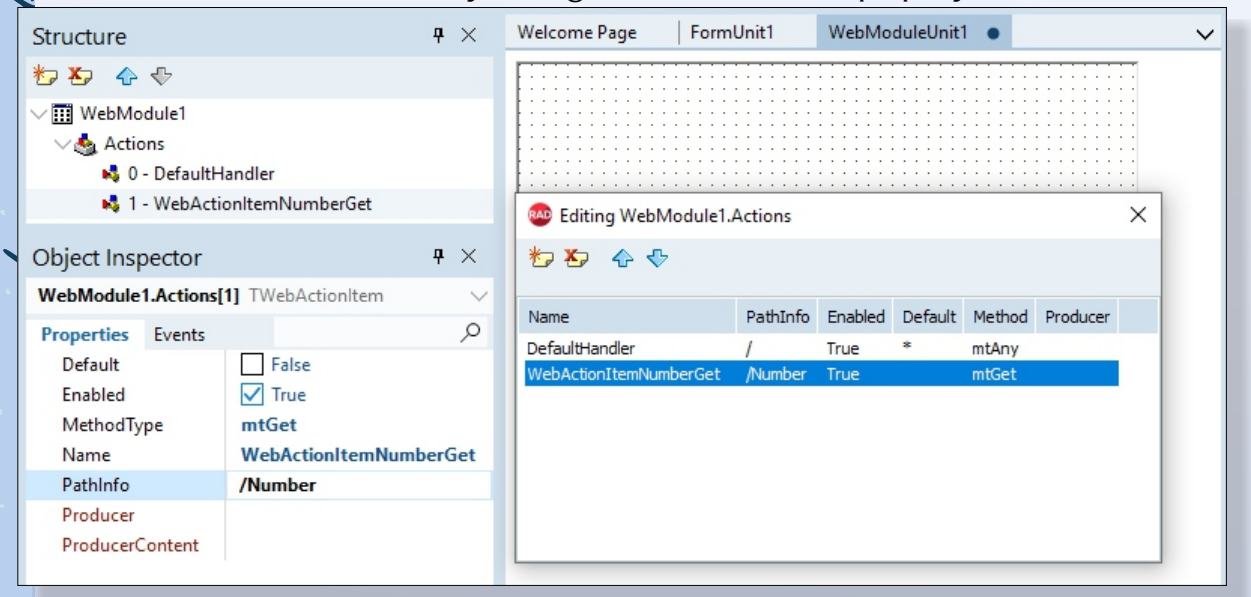


Figure 14:

32. Modify MethodType to mtGet and PathInfo to /Number
33. Add an OnAction event-handler to this WebActionItem and code a response

```
procedure TWebModule1.WebModule1WebActionItemNumberGetAction(Sender:
TObject;
 Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 Response.ContentType := 'application/json; charset=UTF-8';
 Response.Content := Random(100).ToString;
end;
```

34. Run the Web Service Server, and Start it with the Button
35. Now open the Number URL in the Web Service Client
    `http://localhost:8080/Number`
36. Request a couple of random numbers in the Web Service Client

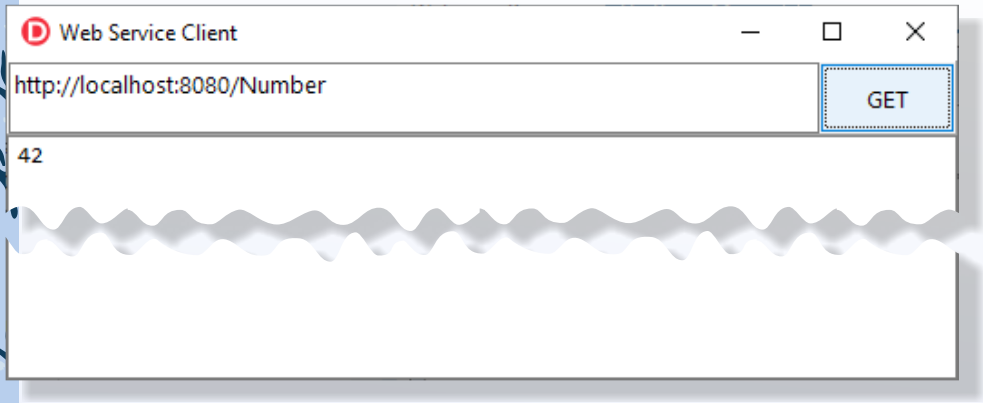


Figure 15: Opening the Number Url

37. By the way, this is valid JSON, because a Number is a primitive datatype is where no encapsulation or serialization is needed.

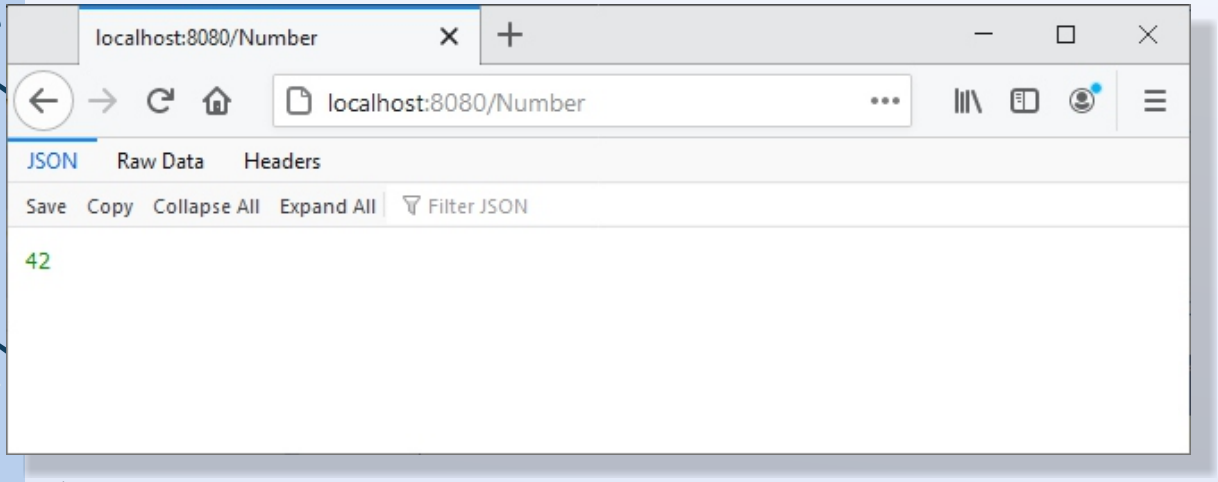


Figure 16:

38. See here also for a brief description of JSON structures
    `https://www.json.org/json-nl.html`

You already have the basis for your own Web Service. You can use this code for example to use the values from your own weather station via GET requests available and displayed in an Android App.

In the following articles, we will change data in the Web Service by adding a PUT, POST and DELETE. We will also use more complex JSON structures.

You can download the source code for this article from your subscription webpage:
`https://www.blaisepascalmagazine.eu/your-downloads/`

WEB SERVICE PART 2 - STORAGE

By Danny Wind

starter ▲ expert

This series of articles is about writing your own web services server and client in Delphi. The approach of all articles is pragmatic. The first article introduced some of the concepts you need to know and shows you how to create and consume your own web service in Delphi with just the GET request. This second article shows you how to update the data in the web service and how to create in-memory storage for the web service.

In the previous article we only used the HTTP GET request to return data from our web service. This time we will add the other three HTTP commands to our web service.

consumes your web service could be caching requests. It's easy to notice when you run into this.

If you get the same result from a GET request, even if the data in the server has changed, then your web client is caching. Taking a look at network traffic helps as well. If the web client only generates network traffic on the first net request, you know what's happening. It's easy to prevent this type of caching behaviour, by setting the Cache-Control and Expires elements in the HTTP header. Remember this if you're not getting the new data that you want to get from your GET.

HTTP GET idempotent, cacheable	Retrieves data from the resource
usage in our web service	**SELECT** (get existing record, disallow caching so we get new data each time)
HTTP POST not idempotent, not cacheable/stale	Appends data to the existing resource
usage in our webservice	**UPDATE** existing (partial update of fields in a record, not updating the primary key)
HTTP PUT idempotent, not cacheable/stale	Replace the existing resource or inserts data as a new resource
usage in our web service	**INSERT** new (or **REPLACE**) (insert new record with new primary key, or replace entire record)
HTTP DELETE idempotent, not cacheable/stale	Deletes the resource
usage in our web service	**DELETE** (delete existing record or return error if it doesn't exist)

Before we do that however we need to know what the idempotent and cacheable properties of the HTTP commands mean for our web service.
Idempotency means that after the first request a second (or third and so on) request of an idempotent method should yield the same effect, unless there is an error or it has expired.

The cacheable property means that a requester is allowed to cache a request. So instead of sending a repeated request to the server again, it can just return a cached result. Both idempotency and cacheable combined in the GET request means that a web client that

IDEMPOTENT METHODS (RFC definition)
Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of N > 0 identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE share this property. Also, the methods OPTIONS and TRACE SHOULD NOT have side effects, and so are inherently idempotent.

However, it is possible that a sequence of several requests is non- idempotent, even if all of the methods executed in that sequence are idempotent. (A sequence is idempotent if a single execution of the entire sequence always yields a result that is not changed by a reexecution of all, or part, of that sequence.) For example, a sequence is non-idempotent if its result depends on a value that is later modified in the same sequence.
A sequence that never has side effects is idempotent, by definition (provided that no concurrent operations are being executed on the same set of resources).

Looking at PUT, you see that PUT is considered idempotent and not-cacheable. So the same PUT request, when repeated, should yield the same result. However PUT is not considered cacheable, so if you PUT a resource, then DELETE that resource from another location and PUT it again it will result in the new resource. The second PUT is not cached on the client-side, it's always considered stale and thus sent to the server. In short, if you PUT something twice it should always successfully replace the existing resource. Your PUTs won't disappear.

For POST the defined behaviour is a bit different, as POST is not idempotent. So if you want to update a resource it could work the first time, but if someone else uses DELETE on that resource a subsequent POST (append) to the same resource could yield an error or just fail.

HTTP Commands
A good thing to know is that the definition of the HTTP commands in RFC allows for multiple usage scenarios of each command.
Because there is some leeway between definition and interpretation of its usage for PUT and POST in web services it's perfectly valid for us to create a web service that uses PUT as an equivalent for INSERT or REPLACE and POST as an equivalent for an UPDATE.

There is an interpretation that wants you to use POST to get data, when the GET request manipulates the data on the server. In our web service we return a random number with a GET request, and this interpretation would suggest using POST instead, as a POST is not idempotent and is allowed to change the server state.

Nevertheless in this article we will continue using GET with Cache-Control and Expires elements in the returned HTTP header to prevent caching.

Before we start coding and add PUT, POST and DELETE methods to our web service, let's first expand our toolkit and introduce the REST Debugger. This is a handy tool that comes bundled with the Delphi IDE. You can use it to test and debug your web service. You can find the REST debugger in the Delphi IDE under Tools ➔ REST Debugger.

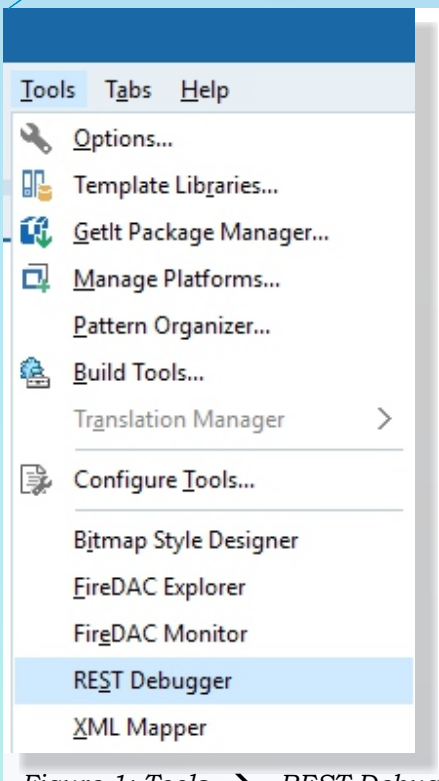


Figure 1: Tools ➔ REST Debugger

Let's run the REST Debugger and use it to test the web service we created in the previous article. We can use the GET request for Number.
`http://localhost:8080/Number`

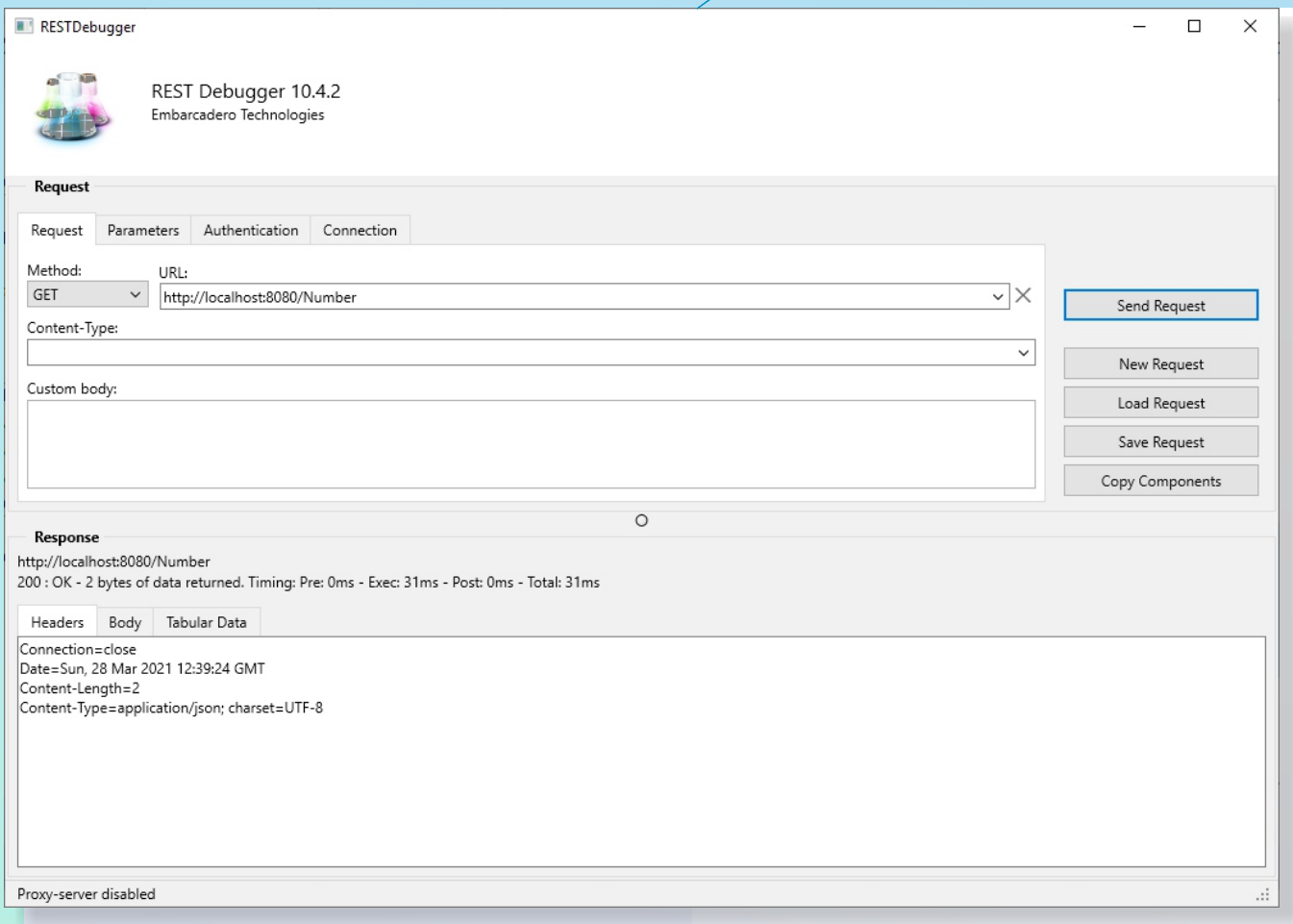


Figure 2:
In the REST Debugger we can see that the returned `Content-Type = application/json` and if we open the tab **Body** we see that the returned JSON is valid.

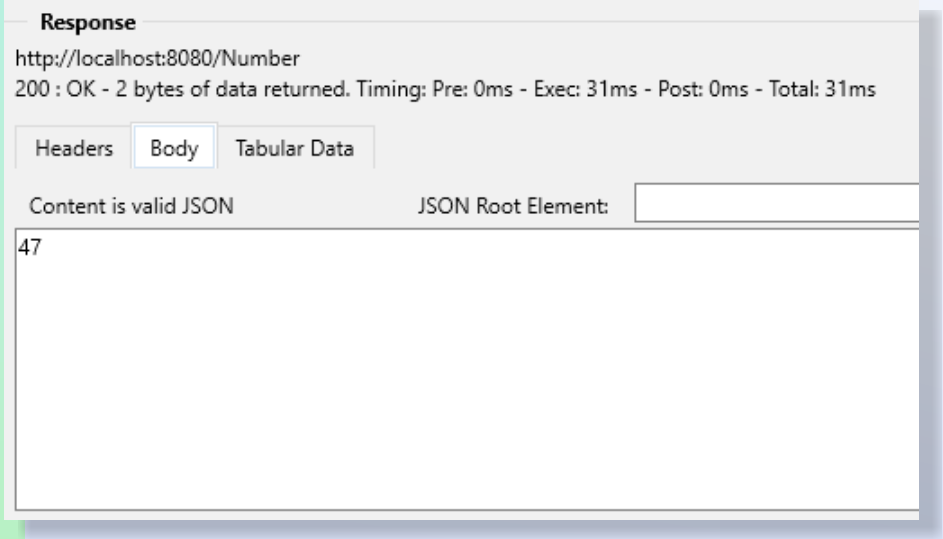
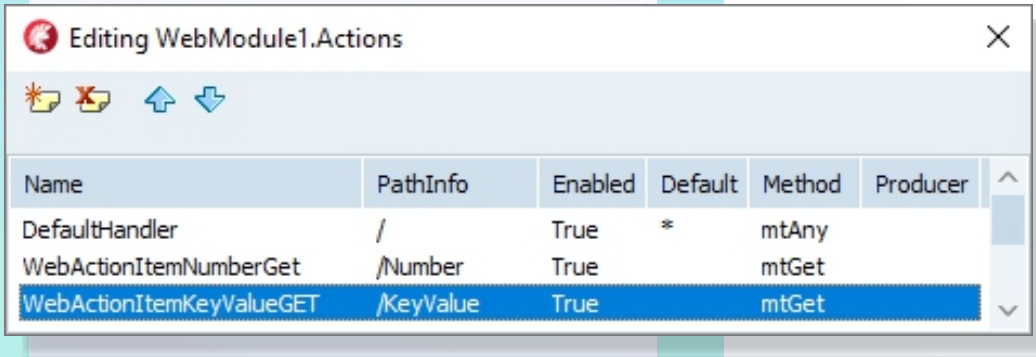


Figure3: Result

We are now ready to add some new methods to our web service.

OPEN THE WEB SERVER SERVICE (**`WebServiceServerWithGUI`**) we created in the previous article in Delphi. It's compatible with either Delphi 10 Community, Delphi 10 Professional, Enterprise or up.

1. In the WebModuleUnit edit the Actions property and add a new handler with name WebActionItemKeyValueGET, MethodType mtGet and pathinfo /KeyValue. *Figure 4:*



2. In the OnAction event-handler for this new item code a test response in the format of a JSON string.

```
procedure TWebModule1.WebModule1WebActionItemKeyValueGETAction(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
begin
  Response.ContentType := 'application/json; charset=UTF-8';
  Response.Content := '{"message":"it works"}';
end;
```

3. Run the web service again, click the Start button and start the REST Debugger and test if your web service still works and a GET request of this URL yields the expected valid JSON result.

`http://localhost:8080/KeyValue`

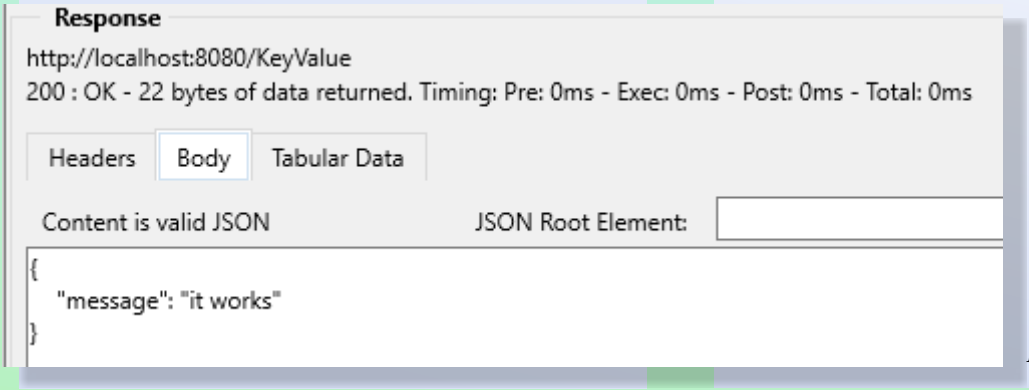


Figure 5: Response

The next step is creating an actual Key Value store in-memory to hold the data for this web service. We will be using a generic TDictionary to store the Key Value pairs.
To safely and successfully use this in-memory Key Value store we need to know how the WebModule handles incoming requests. A Web Broker application has only one WebModule class variable as you can see in the interface section of the WebModuleUnit

```
var
  WebModuleClass: TComponentClass = TWebModule1;
```

However for each request a new WebModule instance may be instantiated and each request is handled in its own thread. For us this means we will need to serialize access to our in-memory Key Value store to make it thread safe. Also the Key Value store will be created as a global variable to make it accessible to all WebModule instances.

4. Declare the gLock and gKeyValueStore variables and add System.Generics.Collections and System.SyncObjs to the uses clause of your WebModule Unit.

```
var  WebModuleClass: TComponentClass = TWebModule1;

implementation

{%CLASSGROUP 'System.Classes.TPersistent'}
{$R *.dfm}

uses
 System.StrUtils, System.Generics.Collections, System.SyncObjs;

var
 gLock: TObject;
 gKeyValueStore: TDictionary<string, string>;
```

❺ At the end of the WebModule unit add an initialization section where you create both the locking object and the Tdictionary.

```
initialization

 gLock := TObject.Create;
 gKeyValueStore := TDictionary<string, string>.Create;
 gKeyValueStore.AddOrSetValue('0', 'Zero');

end.
```

We now have an in-memory Key Value store. To retrieve a value in response from a HTTP GET request we need to do some additional legwork.

First we need to parse the parameters in a HTTP request for our resource identifiers. In a HTTP request to a REST webservice a parameter is usually sent by using additional URL segments. So to get the Value for Key 0 in the KeyValue resource you'd use a URI like this

`http://localhost:8080/KeyValue/0`

For REST web services using URL segments is the preferred and also the most simple method. But if you want to you can also support specifying them as URL query parameters, starting with the question mark and separated by ampersands.

`http://localhost:8080/KeyValue?key=0`

This second method is already supported in the TWebRequest with the QueryFields method, but to support using the preferred URL segment parameters we need to add a bit of code as well as change the PathInfo of the Item for the KeyValue GET.

❻ Modify the PathInfo for the WebActionItem for the KeyValue GET, and add a * at the end like this (see figure 6)

This * makes sure that any URL that starts with /KeyValue, but continues with additional URL segments actually ends up in this WebActionItem handler. So `http://localhost:8080/KeyValue/0` is now also handled by this action handler.

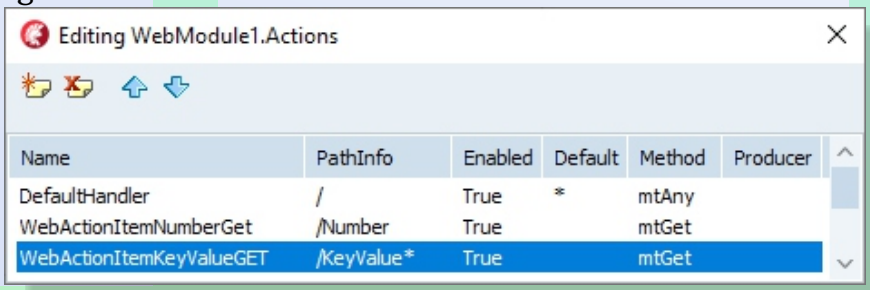
❼ Next we create a function to parse both the URL query parameters as well as the URL segment parameters. Add a protected function declaration to the WebModule.

```
 private
  { Private declarations }
 protected
  function GetParameters(const aActionPath,
aRequestPath: string): TStringDynArray;
 public
  { Public declarations }
 end;
```

❽ And write the following code to parse both types of parameters from the URI

```
function TWebModule1.GetParameters(const aActionPath,
  aRequestPath: string): TStringDynArray;
var
  lActionPathLength, lRequestPathLength: Integer;
  lParameter: string;
  lParameters: TStringDynArray;
begin
  SetLength(Result, 0);
  lActionPathLength := aActionPath.Length;
  lRequestPathLength := aRequestPath.Length;
  if (lRequestPathLength > lActionPathLength) then
  begin
     lParameter := RightStr(aRequestPath,
         lRequestPathLength - lActionPathLength);
     lParameters := SplitString(lParameter,'/');
     if (Length(lParameters) > 0) then
     begin
       Result := lParameters;
     end
  end;
end;
```

Figure 6: Response



❾ Then modify the OnAction event-handler for the WebActionItemKeyValue GET Action

```
procedure TWebModule1.WebModule1WebActionItemKeyValueGETAction(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
  {GET - "Select" / Idempotent}
var
  lParameters: TStringDynArray; lKey, lValue: string;
begin
  lKey := '';
  {parse URL query parameters - http://localhost:8080/KeyValue?key=0}
  lKey := Request.QueryFields.Values['key'];
  if (lKey.IsEmpty) then
  begin {parse URL segment parameters - http://localhost:8080/KeyValue/0}
    lParameters := GetParameters((Sender as TWebActionItem).PathInfo, Request.PathInfo);
    if (Length(lParameters) > 0) then
      begin
        lKey := lParameters[0];
      end;
  end;
  if not(lKey.IsEmpty) then
  begin
    Response.ContentType := 'application/json; charset=UTF-8';
    gKeyValueStore.TryGetValue(lKey, lValue);
    if not(lValue.IsEmpty) then
    begin // {"result":["string"]}
      Response.Content := '{"result":["' + lValue + '"]}';
    end
    else
      begin // {"error":"Item not found"}
        Response.Content := '{"error":"Item not found"}';
      end;
    Handled := True;
    end
    else {No parameters on URL for GET request}
    Handled := False;
end;
```

Test if this works, either with the REST debugger or using the web browser. The results should look like this.

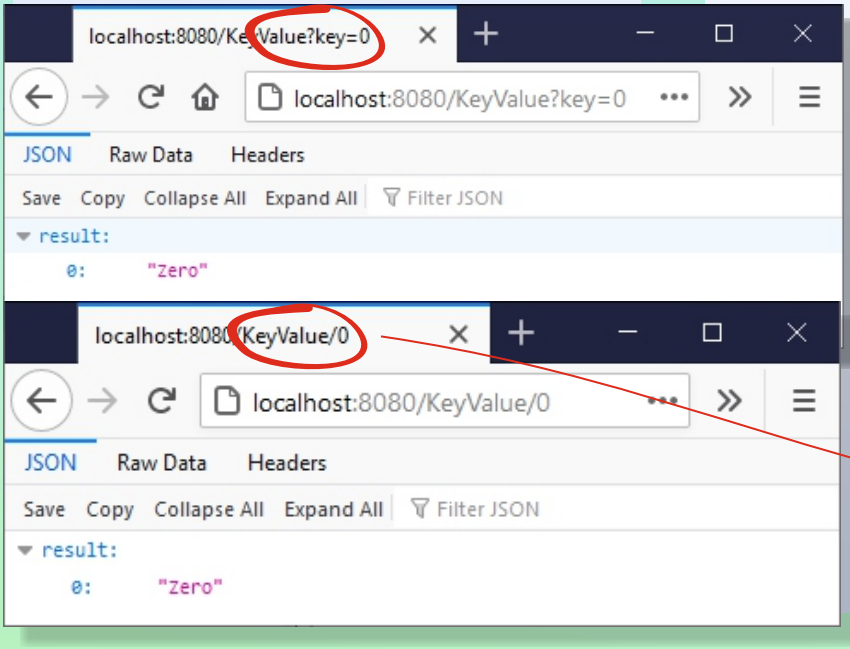


Figure 7: Zero

Our next major step is to add both a PUT and a POST handler.
When data is sent to a web service, the data can be sent as part of a URL segment like this `http://localhost:8080/KeyValue/1/One` but his method has some limitations, one obvious one being that not all characters are allowed in URL segments as they have a special meaning. If you want to send larger or more complex items you would use the HTTP requests body. We want to support both methods of sending data to our web service.

⑩ Add a PUT handler to the WebModule unit, with method type mtPut.

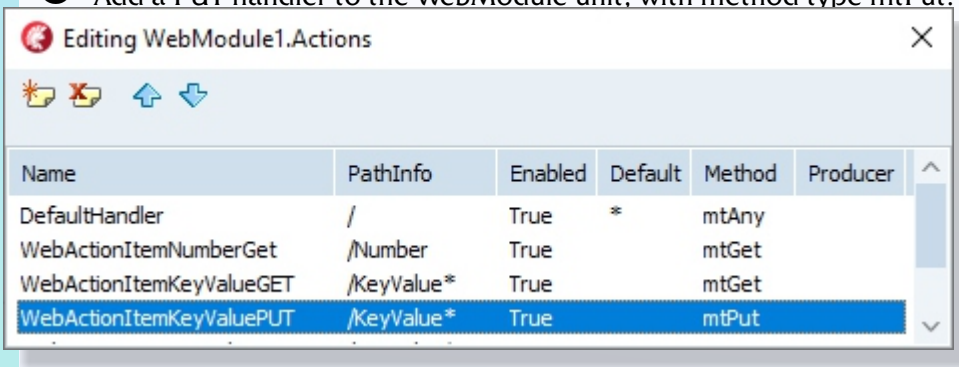


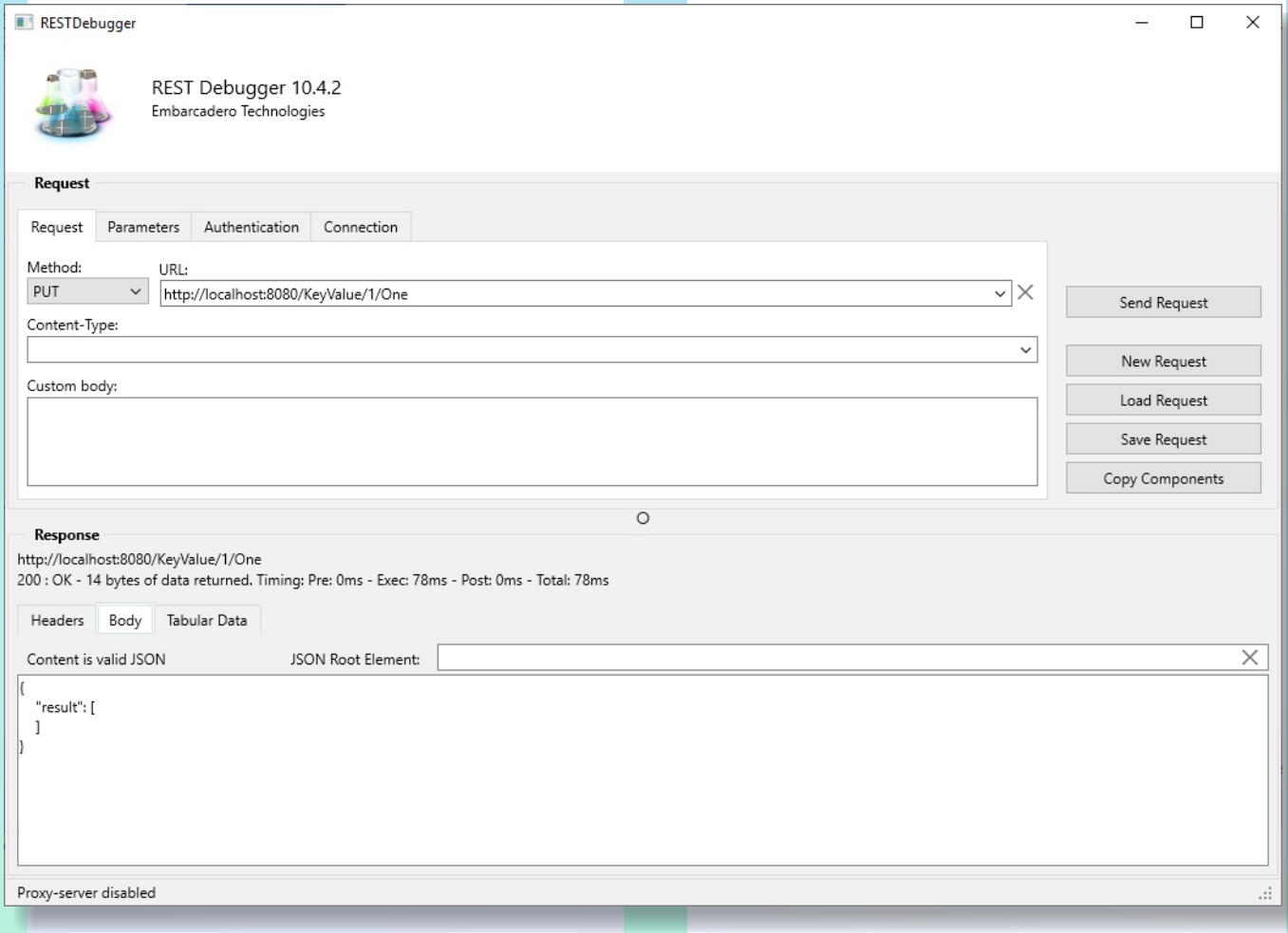
Figure 8: Actions

⑪ Then code the handler

```
procedure TWebModule1.WebModule1WebActionItemKeyValuePUTAction(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
  {PUT - "Insert or Update" / Idempotent}
var
  lParameters: TStringDynArray;
  lKey: string;
  lValue: string;
begin
  lParameters := GetParameters((Sender as TWebActionItem).PathInfo, Request.PathInfo);
  if (Length(lParameters) > 0) then
  begin
    lKey := lParameters[0];
    lValue := '';
    if (Length(lParameters) > 1) then
      begin {Value is part or URL and a simple string}
        lValue := lParameters[1]
      end
  else
    begin {Value is send as content in the request and possibly a JSON or other complex string}
      lValue := Request.Content;
    end;
  if not(lValue.IsEmpty) then
    begin
      Response.ContentType := 'application/json; charset=UTF-8';
      gKeyValueStore.AddOrSetValue(lKey, lValue);
      Response.Content := ' {"result":[]}';
      Handled := True;
    end
  else
    begin
      Handled := False;
    end;
  end
  else
    begin
      {Do not reply}
      Handled := False;
    end;
end;
```

Now test this new PUT method and insert some data into your webservice using the REST Debugger.

Figure 9: Result



And request the new item to see if it was saved correctly.

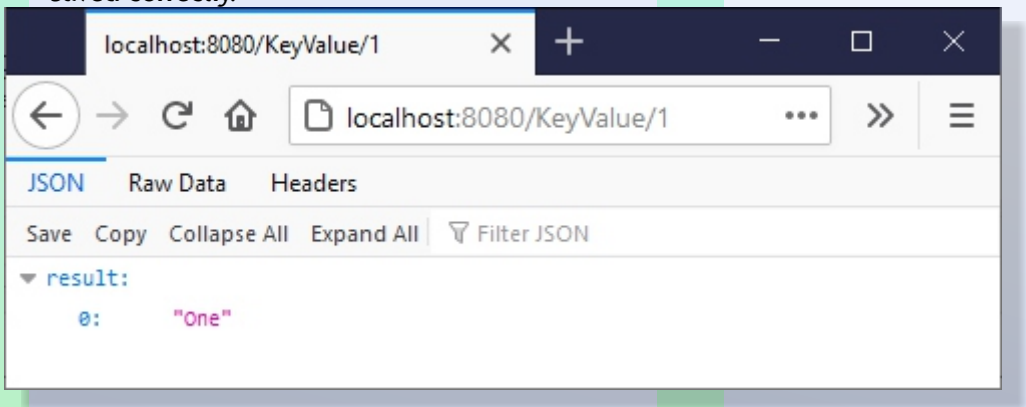


Figure 10: Result One

At first glance the result might seem odd, but that is because the result in this case is returned as a JSON array, and the first item of an array has index 0. If you look at it in the REST debugger you'll see that this is valid JSON.

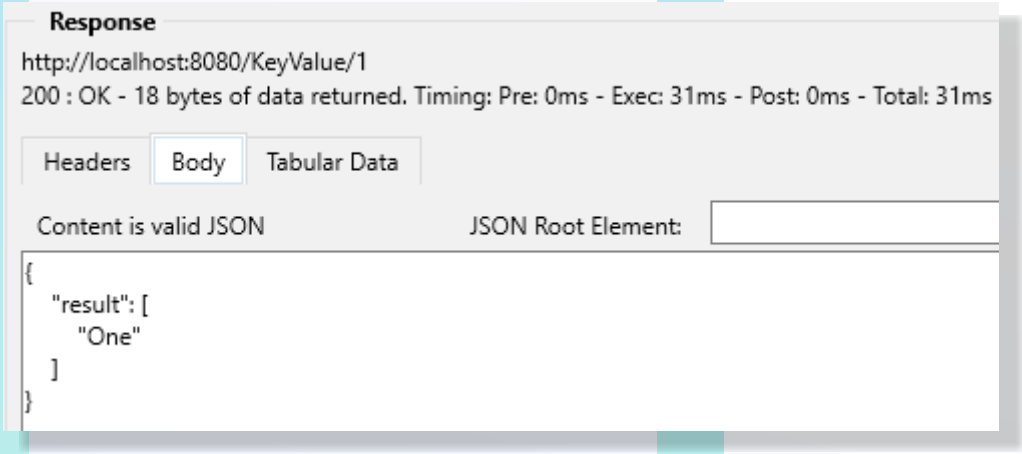


Figure 11: Result One

⓬ We will do the same for POST. We add a handler and use method type mtPOST.

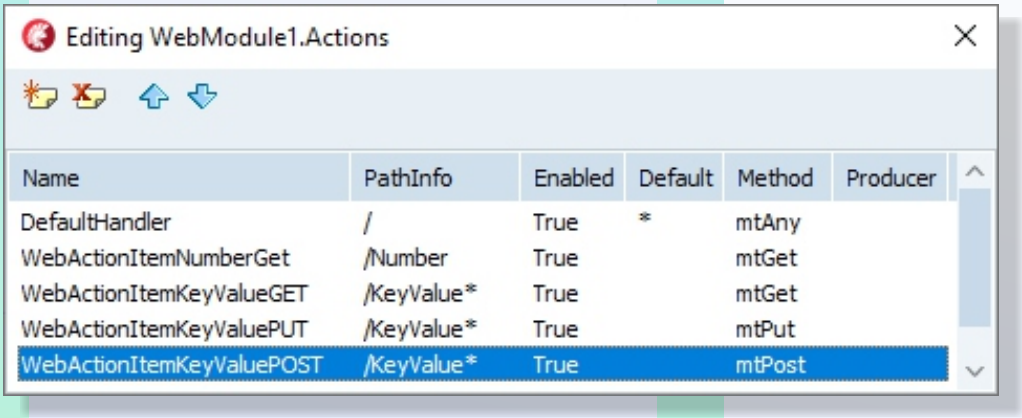


Figure 12: Post

⓭ And the code

```
procedure TWebModule1.WebModule1WebActionItemKeyValuePOSTAction(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
  {POST - "Update"}
var
  lParameters: TStringDynArray;
  lKey: string;
  lValue: string;
begin
  lParameters := GetParameters((Sender as TWebActionItem).PathInfo, Request.PathInfo);
  if (Length(lParameters) > 0) then
   begin
    lKey := lParameters[0];
    lValue := '';
    if (Length(lParameters) > 1) then
      begin {Value is part or URL and a simple string}
        lValue := lParameters[1]
      end
  else
  begin {Value is send as content in the request and possibly a JSON or other complex string}
    lValue := Request.Content;
  end;

  if not(lValue.IsEmpty)
  then
    begin
      Response.ContentType := 'application/json; charset=UTF-8';
      gKeyValueStore[lKey] := lValue;
      Response.Content := ' {"result":[]}';
      Handled := True;
    end
  else
    begin
      Handled := False;
    end;
  end
  else
  begin
    {Do not reply}
     Handled := False;
  end;
end;
```

Test it with the REST Debugger. First add an item with Key 1 and Value One and then update this existing item using POST, changing the Value to something other than One. In my example I used Een, which is the dutch word for One. You probably never guessed you'd learn a dutch word from reading this article.

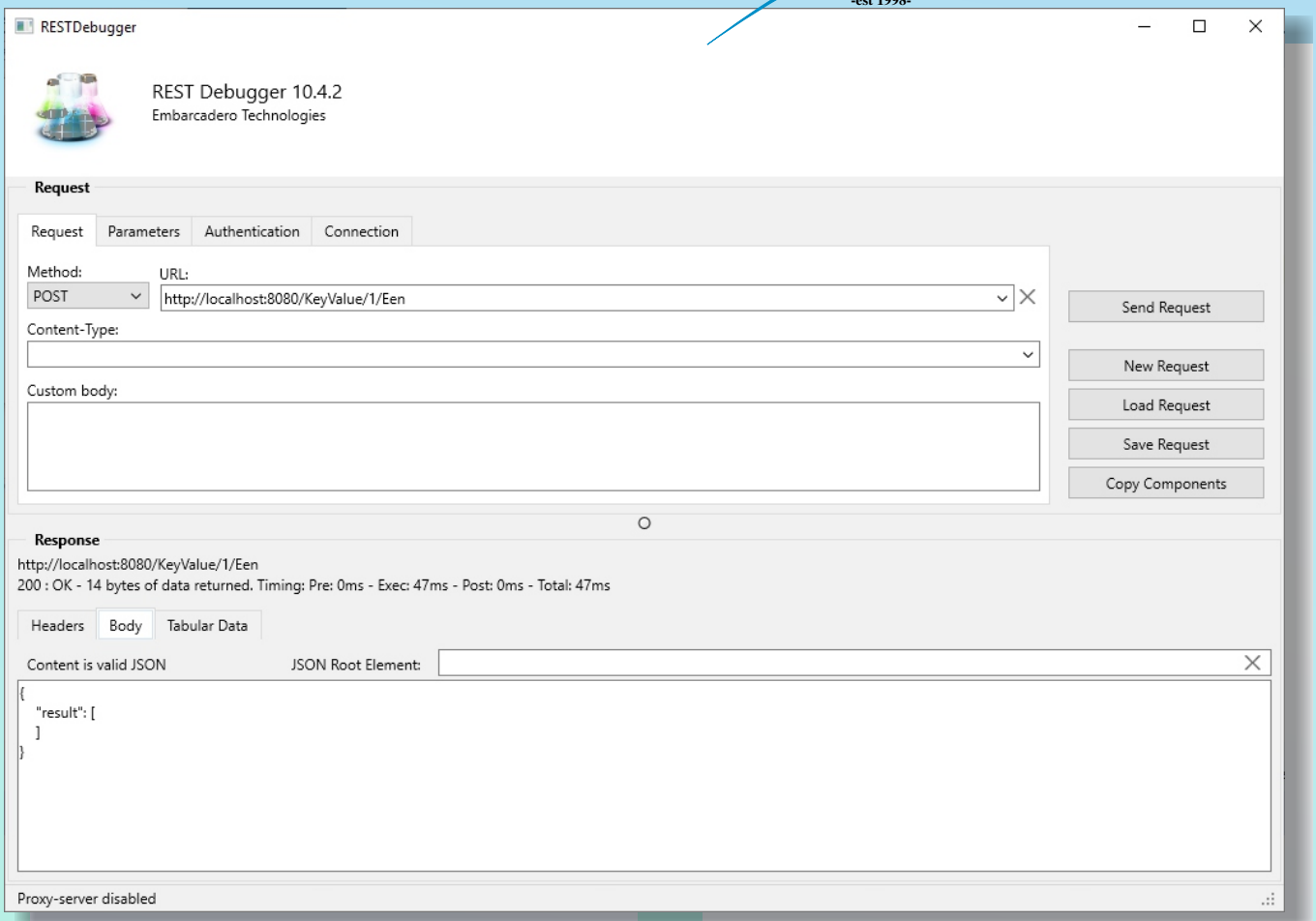


Figure 13: Post

And a request in the web browser to verify it worked.

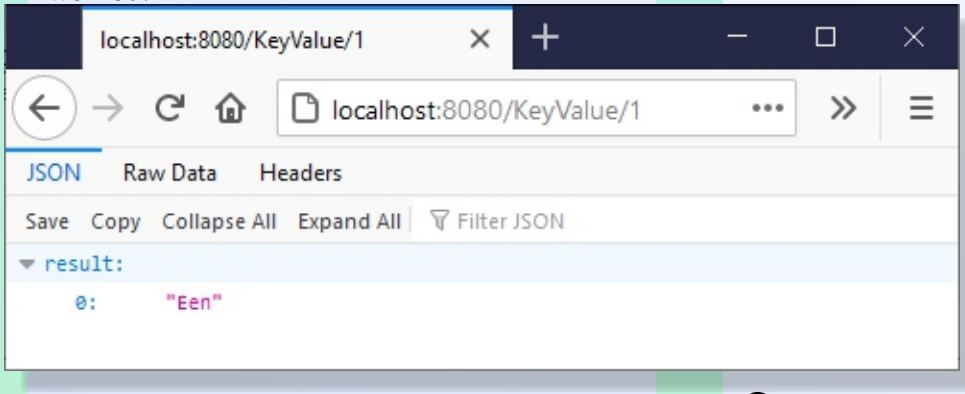


Figure 14: Result: Een

At this point you may have already run into a potential issue we just introduced when adding the `POST` method. What happens if you try to `POST` a Value for a non-existing key?

Just try it out with the REST Debugger.
We need to improve on this, but we will do so in our next article.

14. Our last `HTTP Command` will be the `DELETE`. Just add it to the handlers as before, this time with `mtDELETE`.

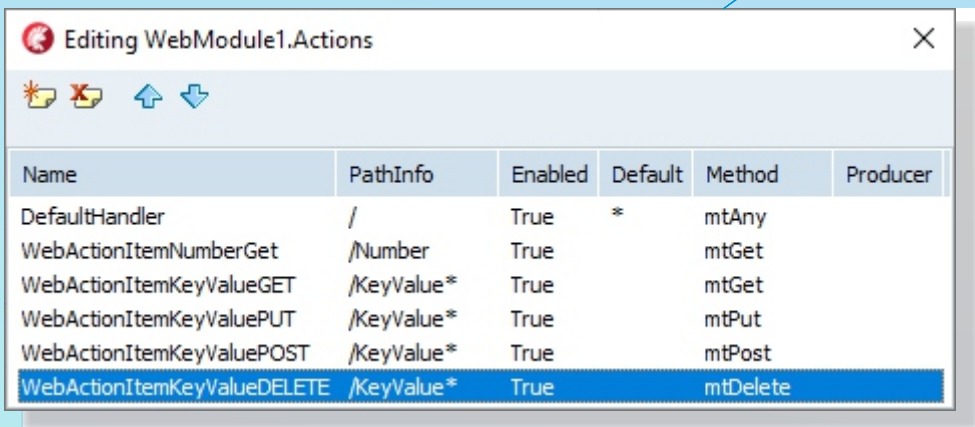


Figure 15: mtDelete

⓯ The code is straightforward:

```
procedure TWebModule1.WebModule1WebActionItemKeyValueDELETEAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
  {DELETE - "Delete"}
var
  lParameters: TStringDynArray;
  lKey: string;
begin
  lParameters := GetParameters((Sender as TWebActionItem).PathInfo, Request.PathInfo);
  if (Length(lParameters) > 0) then
  begin
    lKey := lParameters[0];
    gKeyValueStore.Remove(lKey);
    Response.ContentType := 'application/json; charset=UTF-8';
    Response.Content := '{"result":[]}';
    Handled := True;
  end
  else {No parameters on URL for GET request}
    Handled := False;
end;
```

You may have noticed that we left out URL query parameter parsing in the PUT, POST and DELETE. This was intentional, as using URL segment parameters is the preferred method for a REST web service. However due to the limitations of URL segment parameters, you may wish to add URL query parameters for the PUT/POST and DELETE as well. This could be done quite easily with a duplicate of the sample code from the GET request handler.

A short recap of the things we have done in this article. We created a GET request that correctly parses both URL query parameters and URL segment parameters and returns a value from the in-memory Key Value store. We also created PUT and POST requests that handle passing content as part of the URL segment, for short values, as well as from the HTTP content stream, for larger more complex values.

As a teaser we also added a gLock object, but we did not actually write any code for it. We will do that in our next article, where we'll add some code to make sure any access to our in-memory Key Value store is handled in a thread safe manner. In that next article we will also add error handling and of course expand our Web Service client to consume our web service. There might even be some JavaScript code to consume our web service and further on some additional JSON serialization. Stay tuned!

WEB SERVICE PART 3 - CLIENT

By Danny Wind

starter expert

This series of articles is about writing your own web services server and client in **Delphi.** The approach of all articles is pragmatic.

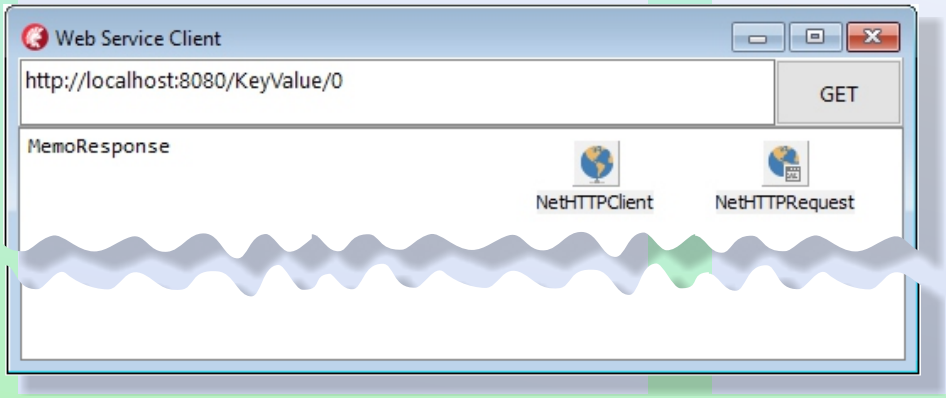
The first article introduced some of the concepts you need to know and shows you how to create and consume your own web service in **Delphi** with just the `GET` request.

The second article showed you how to update the data in the web service and how to create in-memory storage for the web service.

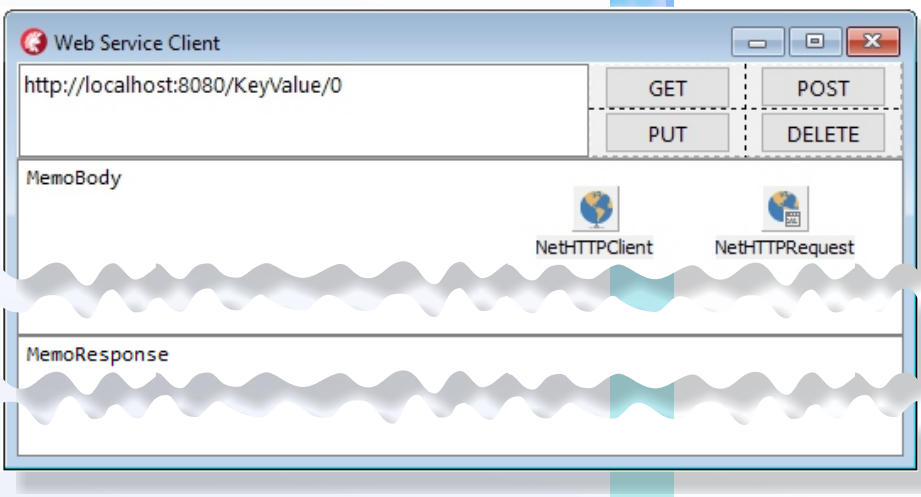
This third article shows you how to consume and use your web service from both **Delphi** clients on **Windows** and from a web page with **JavaScript.**
It also adds error handling and tweaks some code on the web service which were left as teasers in the previous article.

The web service from the previous article is a functional web service that uses the `HTTP` commands `GET`, `POST`, `PUT` and `DELETE` to get, update, insert or delete items in a key value store.
The key value store holds string keys and string values, and can be used to store **JSON** or other string based data.
The **REST** endpoint we defined was
`http://localhost:8080/KeyValue`
and we can `GET` or `DELETE` a value for a given key using parameters in the URL segment.
`http://localhost:8080/KeyValue/0`
Similarly we can `POST` (update existing) or `PUT` (insert or replace) data
`http://localhost:8080/KeyValue/1/One`
but remember that you need to send a `POST` or `PUT HTTP` command, which you can do with the **REST** debugger.
Just opening the above link in a browser would send a `GET HTTP` command.
The `PUT` and `POST` also allow for sending large or complex data within the body of the request instead of using the URL segment.

The Delphi web service client we created in the previous article looked like this



and we will use this web service client as a starting point for our next steps to create a web service client with `POST`, `PUT` and `DELETE` which looks like this:



Open the previous Web Service Client

1. Add a GridPanel under the **GET button** (you can move components around in the **Structure Viewer**, left in the IDE)
2. Set the `Align` of the `ButtonGet` to `None` and the `Anchors` to empty
3. Add three additional Buttons to this **GridPanel** and rename them to `ButtonPost`,`ButtonPut`, `ButtonDelete`
4. Temporarily set `Align` of `MemoResponse` to `None` and move it down
5. Add a Memo to the Form, place it between the `MemoResponse` and the Edit and then align `Top`
6. Set `Align` of MemoResponse back to Client
7. Add an `OnClick` event-handler on the **Button Put** to add the code to **Insert** or **Replace** a value in the web service

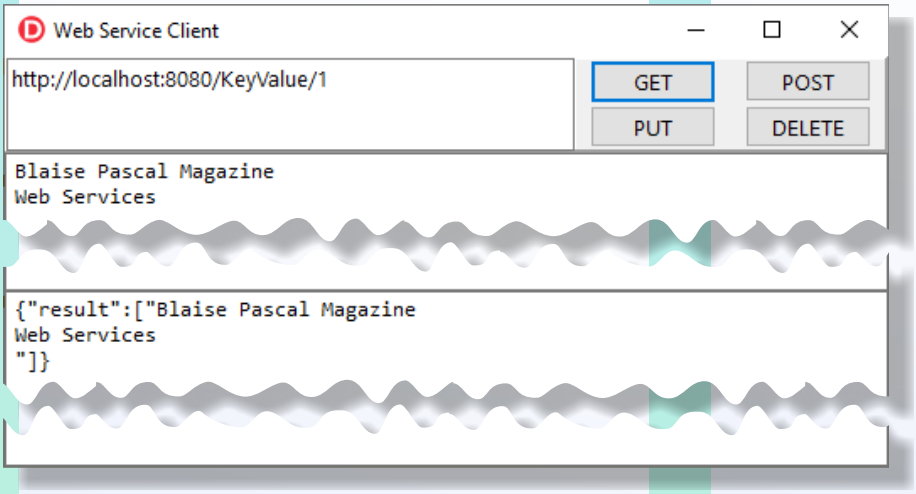
```
procedure TFormMain.ButtonPUTClick(Sender: TObject);
var
 lContentStream: TStringStream;
begin
 { Encode string stream as UTF8 }
 lContentStream := TStringStream.Create(MemoBody.Lines.Text, TEncoding.UTF8);
 lContentStream.Seek(0, TSeekOrigin.soBeginning);
 NetHTTPRequest.Put(EditURL.Text, lContentStream, nil, nil);
end;
```

In this code we use the body of the `HTTP` request to send our data with the `NetHTTPRequest.Put`. We could also have added it as a URL segment parameter in the `EditURL.Text`, but that would have more limitations,

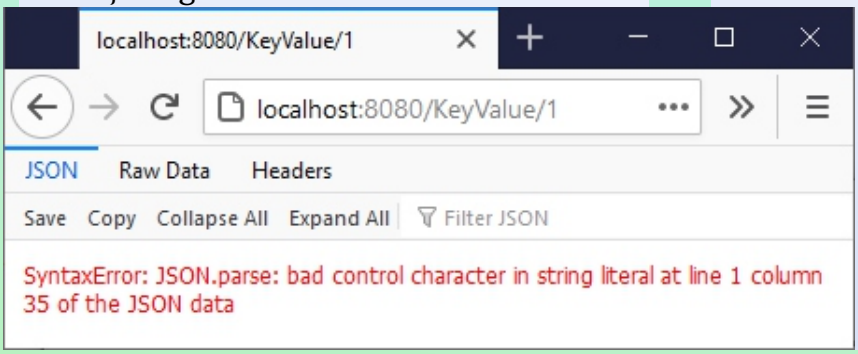
in size and in the supported or allowed characters. Because we use a stream, we also need to manually encode the text from the Memo into `UTF-8` which is the default for sending string data to a web service.
This is also more efficient than the Windows default `UTF-16` encoding, resulting in an up to 50% smaller content.

❽ Test if it works by running the web service server from the previous article. You can also use the completed version of the web service server from this article

❾ Use the following `PUT` URL to place a value in key 1 and then use `GET` to retrieve it. The result in the web service client should look like this



❿ **Notice** how the carriage return - line feeds have also been stored in the **`key value`** store and they result in multiple lines in the **`MemoResponse`**. We should encode these special characters to conform to **JSON** standards to prevent other clients from rejecting our malformed **JSON**



IETF - The Internet Engineering Task Force
specification of the JSON data interchange format states:
"All Unicode characters may be placed within the quotation marks, except for the characters that MUST be escaped: quotation mark, reverse solidus, and the control characters (U+0000 through U+001F)."
Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90,
RFC 8259, DOI 10.17487/RFC8259, December 2017,
<https://www.rfc-editor.org/info/rfc8259>.

⓫ To correctly store a string as a JSON string we need to add JSON string conversion for the control characters and enclose it in quotation marks. We use the function `TJSONString.ToJSON(Options: TJSONOutputOptions)` to convert the string into a JSON string. Modify the code as follows

```
procedure TFormMain.ButtonPUTClick(Sender: TObject);
var
 lContentStream: TStringStream;
 lJSONString: TJSONString;
begin
 { Encode string stream as UTF8 }
 lJSONString := TJSONString.Create(MemoBody.Lines.Text);
 lContentStream := TStringStream.Create(
  lJSONString.ToJSON([TJSONAncestor.TJSONOutputOption.EncodeBelow32]),
  TEncoding.UTF8);
 lJSONString.Free;
 lContentStream.Seek(0, TSeekOrigin.soBeginning);
 NetHTTPRequest.Put(EditURL.Text, lContentStream, nil, nil);
end;
```

With this `ToJSON` function the control characters below `U_001F (32)` are encoded, where some of the special characters such as carriage return and line feed are changed to `\r` and `\n`. Note that I choose to use `ToJSON` with only `EncodeBelow32` specified. I do not want Unicode characters above 127 to be encoded to `\uxxxx`, where `xxxx` is the hexadecimal value of the UTF-16 characters, as that would increase the length of our content. Especially since the latest 2017 ietc specification states that JSON interchange must support all UTF-8 characters and escaping normal UTF-8 characters is not necessary.

⓬ We also need to change a bit of code in the server, als the stored JSON string already has its own quotation characters.
For the GET method we modify the code that returns the JSON array and remove the quotes. We assume that each stored value is valid JSON on its own.

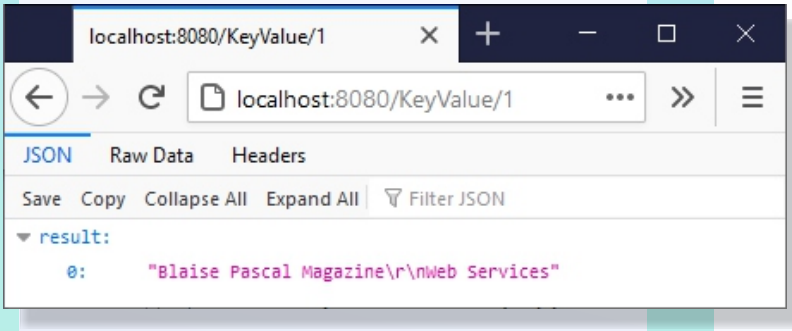
```
Response.Content := '{"result":[' + lValue + ']}'
{ this was Response.Content := '{"result":[''' + lValue + ''']}';}
```

⓭ Also change the test value for key 0

```
gKeyValueStore.AddOrSetValue('0', '"Zero"');
{this was gKeyValueStore.AddOrSetValue('0', 'Zero');}
```

This is not totally foolproof, as it assumes anyone pushing data into the key value store adds valid JSON, but it's good enough for our simple web service.

⓮ If we now test the service by storing the two lines we get this correct result in the browser



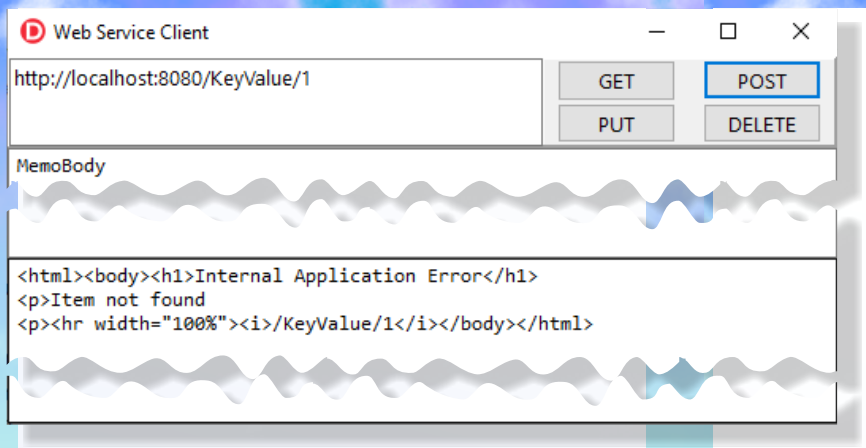
⓯ All looks OK, however if this looks strange to you, remember that we return a JSON array of values with one item (0) with the value for key 1

⓰ Back to the client

⓱ The following code implements the POST functionality

```
procedure TFormMain.ButtonPOSTClick(Sender: TObject);
var
 lContentStream: TStringStream;  lJSONString: TJSONString;
begin
 { Encode string stream as UTF8 }
 lJSONString := TJSONString.Create(MemoBody.Lines.Text);
 lContentStream := TStringStream.Create(
  lJSONString.ToJSON([TJSONAncestor.TJSONOutputOption.EncodeBelow32]),
  TEncoding.UTF8);
 lJSONString.Free;
 lContentStream.Seek(0, TSeekOrigin.soBeginning);
 NetHTTPRequest.Post(EditURL.Text, lContentStream, nil, nil);
end;
```

⓲ The code is the same as the PUT, with one additional condition that a POST to a non-existent key will fail with an internal error. Note that the web service server neatly translates such an internal exception to a HTML page



⑲ Instead of this **HTML** page I'd like it to return a **JSON** error

⑳ Open the web service server and find the code that handles the **POST** in the **Web Module** unit

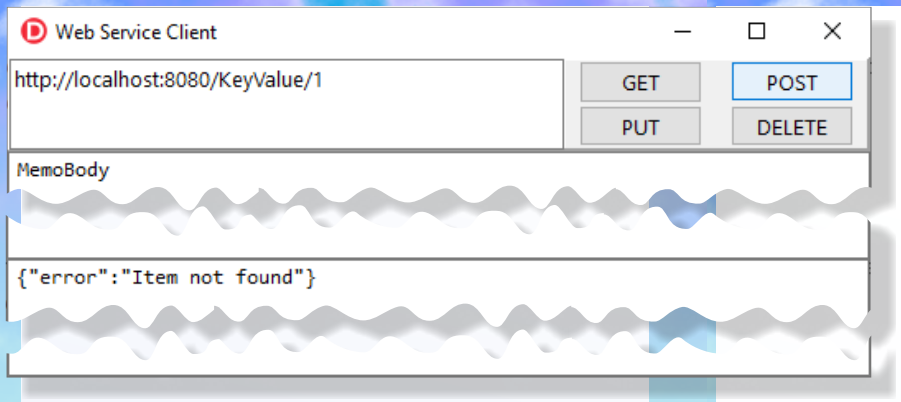
```
{existing code}
procedure TWebModule1.WebModule1WebActionItemKeyValuePOSTAction ...
 ...
 Response.ContentType := 'application/json; charset=UTF-8';
 gKeyValueStore[lKey] := lValue;
 Response.Content := '{"result":[]}';
 Handled := True;
```

㉑ and modify it to return a **JSON** formatted error string if the key is not found in the key value store

```
new code}
procedure TWebModule1.WebModule1WebActionItemKeyValuePOSTAction ...
 ...
 Response.ContentType := 'application/json; charset=UTF-8';
 if gKeyValueStore.ContainsKey(lKey) then
 begin
  gKeyValueStore[lKey] := lValue;
  Response.Content := '{"result":["OK"]}';
 end
 else
 begin
  Response.Content := '{"error":"Item not found"}';
 end;
 Handled := True;
```

㉒ and instead of returning an empty **JSON** array, we now also return one array item with "OK", making it easier to parse

㉓ After this code change the result after a click on POST with a non-existent key should look like this



24. We go back to the web service client and we finish the client side code with the DELETE

```
procedure TFormMain.ButtonDELETEClick(Sender: TObject);
begin
 NetHTTPRequest.Delete(EditURL.Text, nil, nil);
end;
```

25. After which we have a fully functional web services client

The web services client adds values as **JSON** strings, the web services server stores these as-is and when requested returns the **JSON** value as the first item in a **JSON** array.

Maybe at this point you are wondering why we use a **JSON** array to return just one item.
That is because using an array is a flexible way of returning items with **JSON.** We can use the **JSON** iterator in a later article to parse for multiple items, for instance if we request the entire list, or if we want to return additional items that describe the content of the value for each key. We could put a value in the key value store that is actually a **BSON** encoded binary file with a descriptor that holds the file type and return the descriptor, which could be a **MIME** type, as an item as well.

On the subject of **MIME** types, there is a small improvement you could make to the header that is sent out by the server. It is currently just a manual string

```
Response.ContentType := 'application/json; charset=UTF-8';
```

but you could change it to

```
Response.ContentType := 'application/json; charset=' + TEncoding.UTF8.MIMEName;
```

This would result in almost the same string, but `UTF-8` would now be written in lower case.

```
'application/json; charset=utf-8'
```

Although using upper case is allowed, as the charset specification is case-insensitive, the default should be in lower case. I just made this mistake when typing the article, as in normal text I tend to use `UTF-8`.
Using `TEncoding.UTF8.MIMEName` instead makes sure I don't repeat that same mistake.

Another thing I forgot to mention was setting the `TNetHTTPRequest` property `Asynchronous` to `True` (default is False) in the web services client. The code also works in synchronous mode, but it is meant to be used asynchronously.

We also have some other things to do that we didn't get around to in the previous article. Let's revisit some code on the server side.

In our previous article we declared and created a global lock variable, but we did not actually use it. If you have Show Error Insight levels set to **"Everything"** under **Tools-Options** in **Delphi** 10.4.2 you'll get a visual indication the code is incomplete if you open the `WebModule` unit

```
initialization

  gLock := TObject.Create;
  gKeyValueStore := TDictionary<string, string>.Create;
  gKeyValueStore.AddOrSetValue('0', 'Zero');

end.
```

H2077 Value assigned to 'gLock' never used

We will add this code soon, but first we dig into the reason why we need to add a global lock.
You may recall that a Web Broker application only has one WebModule class variable as you can see in the interface section of the WebModule unit

```
var
WebModuleClass: TComponentClass = TWebModule1;
```

However for each request a new WebModule instance of this WebModuleClass type may be instantiated and each incoming request is handled in its own thread.

Instantiation of **WebModules** is handled by the `WebRequestHandler`. **WebModule** instances (of the `WebModuleClass`) are kept in a pool in the handler, if one instance is available the `WebRequestHandler` will use that one, if not a new one will be created.
Threading is handled by the Indy `HTTP` Server. By default the `IdHTTPServer` handles each request by creating its own new thread. If we would create the web service server as `ISAPI` or Apache the threading would be handled there.

For us knowing that we have multiple instances of **WebModules** used from multiple threads at the same time, means we will need to serialize access to our one global in-memory `Key Value` store to make it thread safe. This is where we will use the global lock variable `gLock` as a companion lock object for the `TDictionary` in combination with `Tmonitor`.

`TMonitor` is an excellent choice for locking in multi-threaded applications. Internally `TMonitor` first uses spin waits before actually locking, which reduces context switching.
The lock flag is also built into each class in Delphi through the `TObject` base class.
When locking an object it's good practice to use a companion `TObject` instance, instead of just locking the class directly.
This is because for some classes in the **Delphi RTL** `TMonitor` is also used in its internal code. Using `TMonitor` on such a class could lead to deadlocks. Instead just declare a new `TObject` variable, as we do in our code with gLock, to lock access to the key value `TDictionary`.

26. In each of the methods that access the `Key Value` store we add a lock by surrounding it with `TMonitor.Enter` and `Exit`. For the `GET` web action item handler the new code looks like this

```
Response.ContentType :=
 'application/json; charset=' + TEncoding.UTF8.MIMEName;
if TMonitor.Enter(gLock, 500) then
begin
 try
  gKeyValueStore.TryGetValue(lKey, lValue);
 finally
  TMonitor.Exit(gLock);
 end;
end;
if not(lValue.IsEmpty) then
begin
 // {"result":[JSONValue]}
 Response.Content := '{"result":[' + lValue + ']}'
end
else
begin
 // {"error":"Item not found"}
 Response.Content := '{"error":"Item not found"}';
end;
Handled := True;
```

The **TMonitor.Enter** has a timeout parameter, if the lock is not acquired within 500 milliseconds it will return **False** and the **TryGetValue** will not be executed. Usually the lock will be acquired within < 1 ms, but if the **Key Value** store is busy from multiple threads it may take longer and we do not want to wait indefinitely. Instead getting value will then fail and return a **JSON** error with Item not found. Alternatively you could also handle this with **HTTP** error codes as some web services do.

27 We add similar code for the **DELETE** web action item handler.

```
Response.ContentType := 'application/json; charset=' + TEncoding.UTF8.MIMEName;
if TMonitor.Enter(gLock, 500) then
begin
  try
    gKeyValueStore.Remove(lKey);
  finally
    TMonitor.Exit(gLock);
  end;
end;
```

28 and the **PUT** web action handler

```
Response.ContentType :=
 'application/json; charset=' + TEncoding.UTF8.MIMEName;
if TMonitor.Enter(gLock, 500) then
begin
  try
    gKeyValueStore.AddOrSetValue(lKey, lValue);
  finally
    TMonitor.Exit(gLock);
  end;
end;
```

29 and the POST web action item handler

```
Response.ContentType :=
 'application/json; charset=' + TEncoding.UTF8.MIMEName;
if TMonitor.Enter(gLock, 500) then
begin
 try
  if gKeyValueStore.ContainsKey(lKey) then
  begin
   gKeyValueStore[lKey] := lValue;
   Response.Content := '{"result":["OK"]}';
  end
  else
  begin
   Response.Content := '{"error":"Item not found"}';
  end;
 finally
  TMonitor.Exit(gLock);
 end;
end;
Handled := True;
```

30 After which we have a fully functional web services server

This web services server does have some limitations. Because it is using a globally locked key value store its performance will suffer as we get more simultaneous users. If they mostly just `GET` data the penalty for global locking is low as getting data out of a dictionary based key value store is a O(1) operation. It is very quick. However inserting `(PUT)` or deleting `(DELETE)` data from the key value store is somewhat slow as it needs to (re)calculate hash values. If you have many concurrent users that also write a lot I would not use this setup, but instead just use a fast database backend. Using a database backend has the added benefit of persistence.
The current key value store holds values in memory, after a reset of the web service the data is gone. For simple web services that need this type of transient storage this approach works fine.

It's time to have some fun with our web services server. Let's add some **JavaScript** to the mix.

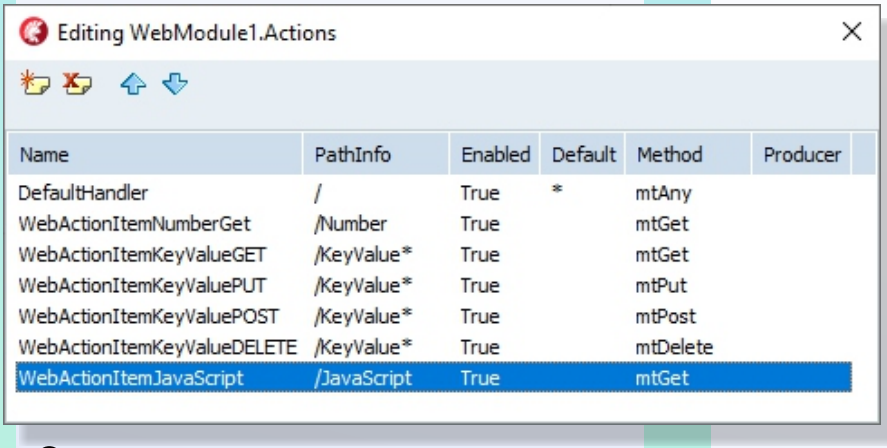
In a previous article I wrote that a web service is not that much different from serving web pages from a web server.
In fact you can add web page producers to the web service server we just wrote and have it return a `HTML` page.
We have already seen that when it returned an internal exception as a `HTML` page.

The default handler in the **Web Module** unit does the same thing,it just returns some `HTML`.

This means that we could add an URL to the web service server that would result in a webpage with some `HTML` and a piece of **JavaScript** that would in turn request data from the same web service.

Kind of like a roundtrip, where the web service asks itself a question. This way we would let the web services server serve a web page that acts like a **JavaScript** client to the same web service.

31. We add a new **WebActionItem** handler to the **WebModule** unit, use the **URL /JavaScript** and the method `mtGet`



Name	PathInfo	Enabled	Default	Method	Producer
DefaultHandler	/	True	*	mtAny	
WebActionItemNumberGet	/Number	True		mtGet	
WebActionItemKeyValueGET	/KeyValue*	True		mtGet	
WebActionItemKeyValuePUT	/KeyValue*	True		mtPut	
WebActionItemKeyValuePOST	/KeyValue*	True		mtPost	
WebActionItemKeyValueDELETE	/KeyValue*	True		mtDelete	
WebActionItemJavaScript	/JavaScript	True		mtGet	

32. In the handler we respond with a piece of `HTML` with **JavaScript** code

```
<html>
<head><title>Call Number with JavaScript</title></head>
<body>Call Number with JavaScript
<button onclick="getNumber()">
Get Number in Console Log (view Ctrl-Shift-I).</button>
<script type="text/javascript">
function getNumber()
{ let url = 'http://localhost:8080/Number';
  fetch(url).then(resp=> resp.json().then(j=>
          console.log('\nNumber: ', j)));
}
</script>
</body>
</html>
```

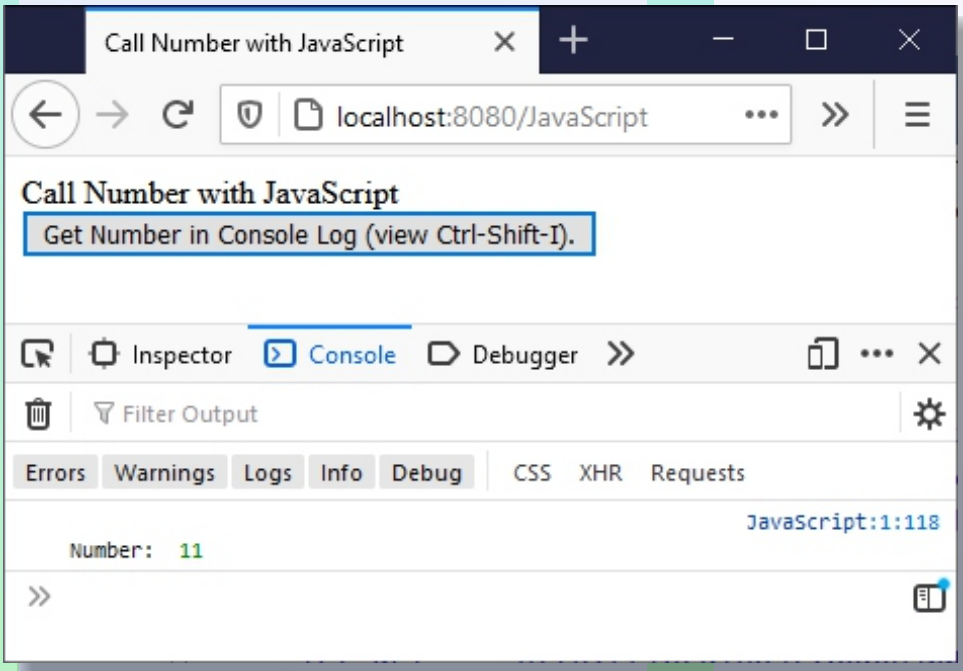
33 The resulting Delphi code is this

```
procedure TWebModule1.WebModule1WebActionItemJavaScriptAction
 (Sender: TObject;Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);
begin
 Response.ContentType :=
  'text/html; charset=' + TEncoding.UTF8.MIMEName;
 Response.Content :=
  '<html>' +
  '<head><title>Call Number with JavaScript</title></head>' +
  '<body>Call Number with JavaScript ' +
  '<button onclick="getNumber()">Get Number in Console Log (view Ctrl-Shift-I).</button>' +
  '<script type="text/javascript">' +
  'function getNumber() {' +
  ' let url = ''http://localhost:8080/Number'';' +
  ' fetch(url).then(resp=> resp.json().then(j=> console.log(''\nNumber: '', j)));' +
  '}' +
  '</script>' +
  '</body>' +
  '</html>';
end;
```

34 For web debugging I usually use either **Firefox** or **Chrome,** you can start the web debugging with the key combination **Ctrl-Shift-I**

35 Run the web services server, click the Start button, then the Browser button and open the JavaScript URL **`http://localhost:8080/JavaScript`**

36 The result after clicking the **JavaScript** button on the page would look like this



37 The output of clicking the **Get Number** button is only viewable in the Console view in the web debugger (**Ctrl-Shift-I**)

The web service server we made can be used from any other platform that supports web services, from **JavaScript, Python** or **PHP** or any other language or platform.
This makes it a simple solution to enable sharing data from your **Delphi** application with other third-party solutions.

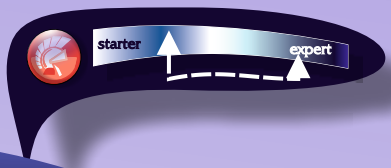
If you start using your **Delphi** web service from other platforms you may run into caching issues and having to configure **Cross-Origin Resource Sharing.**
We will look into these issues in our next article on deployment of the web service server.

A short recap of the things we have done in this article. We created a `POST`, `PUT` and `DELETE` request to the web services client. We also modified the data to be passed as a **JSON** string adding explicit **UTF-8** encoding of the body content that is sent over the network to the web services server. We added thread-safety code to the web service server and to top it all off we added a `HTML` page with some **JavaScript** code that does a roundtrip and asks our web service for a random number.

In our next articles, we will take a look at **ISAPI** and **Apache** versions of our web services server and also how to deploy each of these to a server and allow access to our web service. Along the way we will tweak some settings to improve interoperability, with `HTTP` headers for caching and **CORS** and we will configure our web service for better performance. We may also add some more **JSON** support, parsing the result array and adding serialization of objects or even use the web service to store other items besides plain text.
Maybe that last bit will be in another article though. Stay tuned!

WEB SERVICE PART 4: DEPLOY TO INTERNET INFORMATION SERVICES

By Danny Wind



This series of articles is about writing your own web services server and client in Delphi. The approach of all articles is pragmatic. The first article introduced some of the concepts you need to know and shows you how to create and consume your own web service in Delphi with just the GET request. The second article showed you how to update the data in the web service and how to create in-memory storage for the web service. The third article showed how to consume and use your web service from both Delphi clients on Windows and from a web page with JavaScript. This fourth article is about deploying your web service to the Internet Information Services (IIS) server on Windows.

What are the deployment options for a web service written in Delphi? If you create a new web application in Delphi you have the following options in the wizard

Apache dynamic link module	An Apache module. Apache has support for HTTP and HTTPS. The current Apache 2.4 is supported on x64 Linux.
Stand-alone console application	A stand-alone WebBroker console application is a web server that has a text-only user interface. It supports HTTP using an Indy HTTP server component.
Stand-alone GUI application	A stand-alone WebBroker application is a web server that displays a form. It supports HTTP using an Indy HTTP server component.
ISAPI dynamic link library	An ISAPI library integrates with IIS. IIS has support for HTTP and HTTPS.
CGI stand-alone executable	A CGI executable integrates with a web server. Note that CGI is typically slower and more difficult to debug than ISAPI or an Apache module.

The stand-alone options do not use the IIS or Apache web server platform for receiving and handling HTTP requests, but instead rely on the Indy HTTP web server component. This works reliably and can even support HTTPS with OpenSSL, although the wizard suggests otherwise. However it is less suited for server environments exposed to the public internet, which need to be maintained, updated and monitored to remain secure. Stand-alone is more suited for static installations hidden inside a VPN or for embedded industrial installations.

One stand-alone option is missing from this list; it's also possible to manually create a Windows Service application and embed the web service into it. This uses the same Indy web server as the other Stand-alone options.

Of these options ISAPI and Apache make the most sense for regular deployment. Both ISAPI and Apache can be run on default web server installations of Windows and Linux. You can just order a Virtual Private Server with any hosting provider and have your web service up and running on the internet within a couple of hours. Of course you can also use your own server. Because IIS and Apache are widely supported you can set up monitoring on the web server with one of the many available tools or delegate maintenance and administration to a third-party system administrator. System administrators are usually well versed in handling security, updates and monitoring of IIS and Apache installations. Both IIS and Apache also support HTTPS encryption, made easy through Let's Encrypt certificates or, if you need, higher level certificates from one of the other Certificate Authorities.

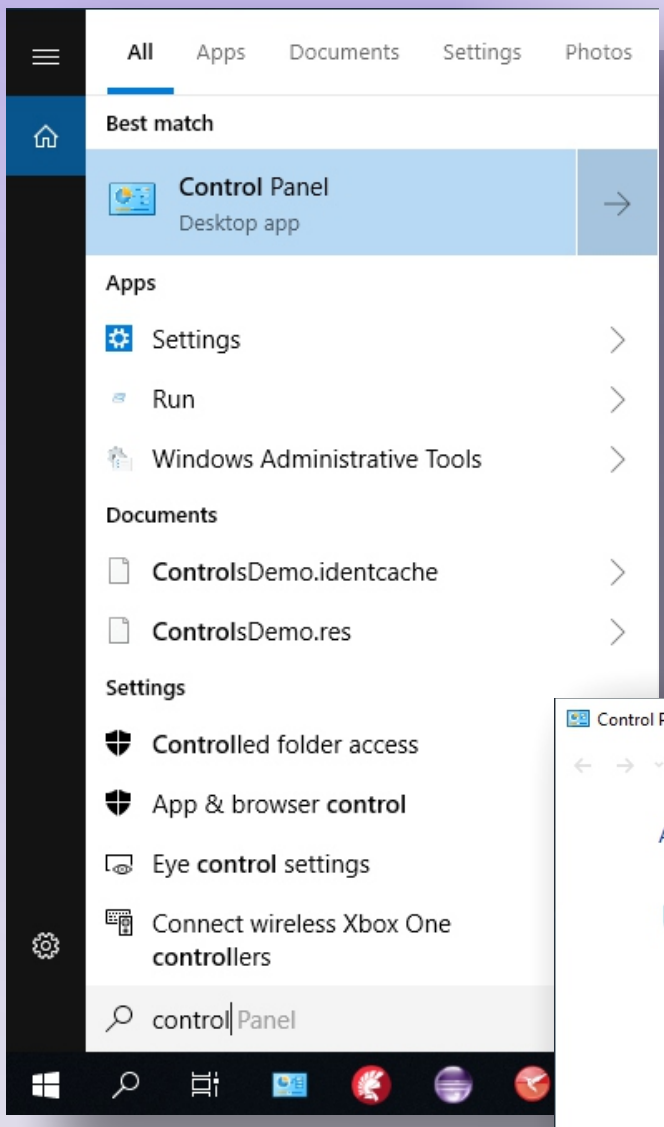
In this article we will focus on using IIS and ISAPI for our web service and we will deploy it to a development machine.

We start by enabling Internet Information Server on a plain Windows-10 machine. This could be the same (virtual) machine where you have your Delphi development installed, but any other Windows-10 machine or virtualized environment is fine as well.

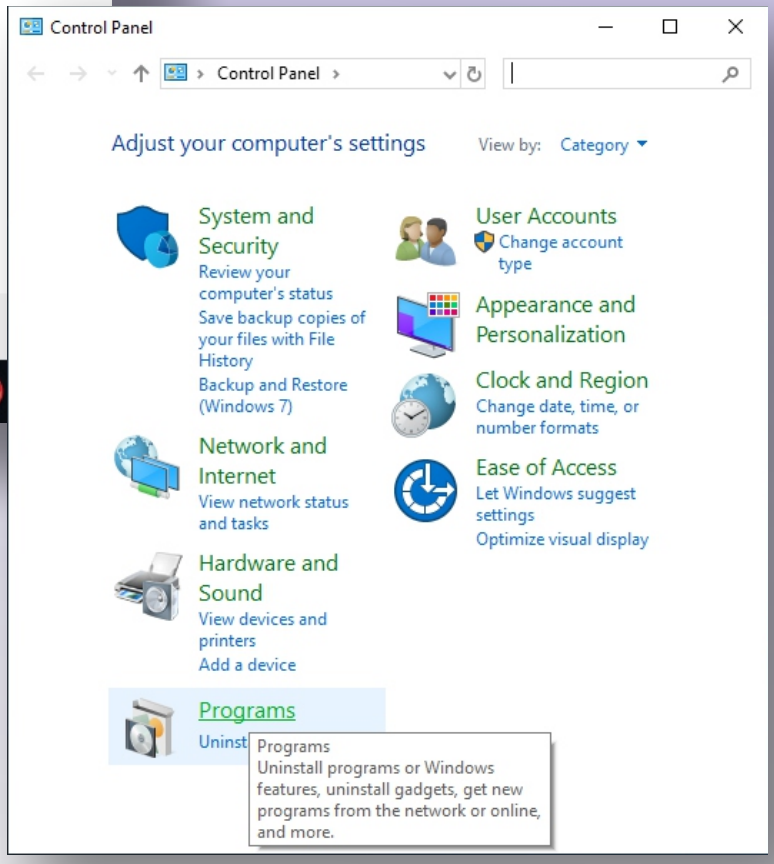
If you are running a Windows Server environment you probably already have Internet Information Services installed through your server roles administration panel. In that case you can skip ahead towards creating the ISAPI itself in Delphi at step 7.

For those of us who are using plain Windows-10 we install the Internet Information Services server through the Control Panel with Programs and Features

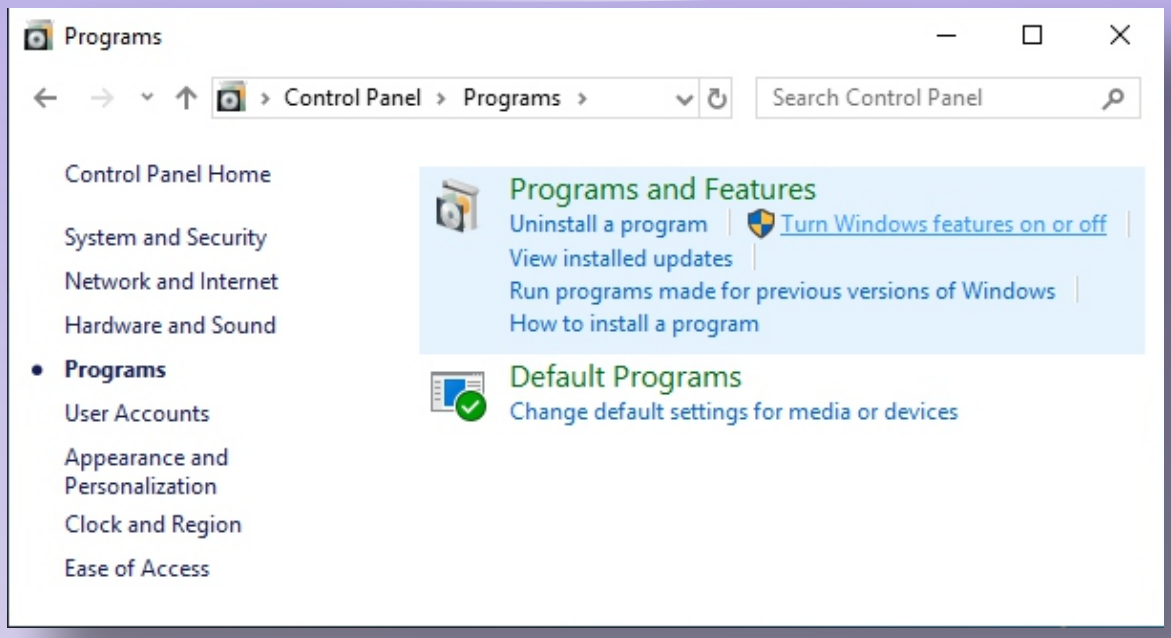
❶ Click on the search icon next to the start menu and type “Control”. Note that even if you are using a non English language version of Windows this will still lead you to the Control Panel Desktop app.



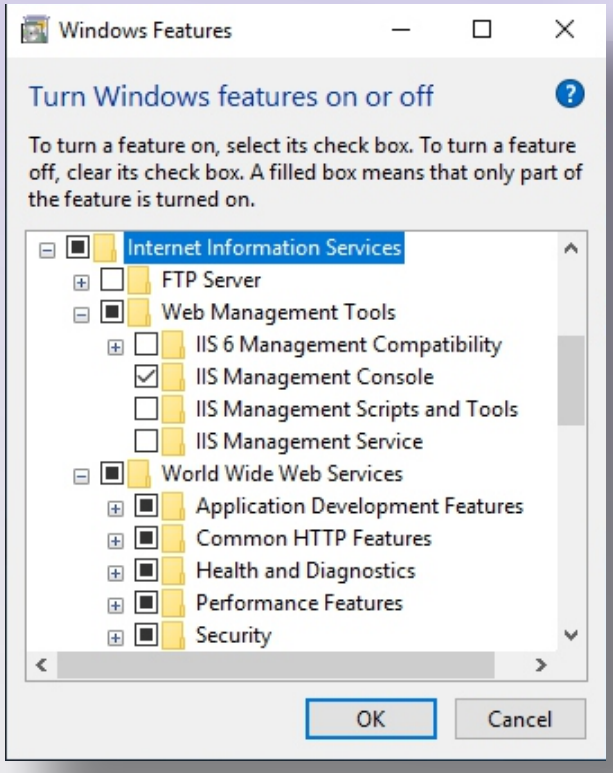
❷ In the Control Panel select “Programs”



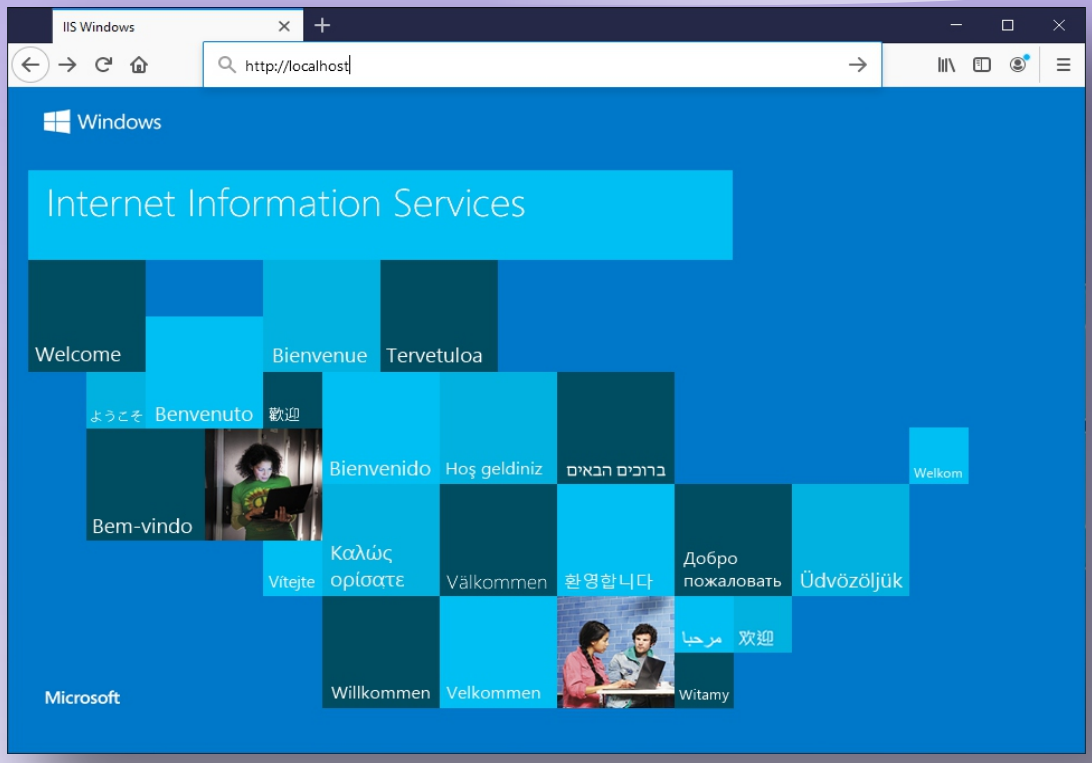
❸ Under Programs click on "Turn Windows features on or off"



❹ Inside the Turn Windows Features on or off selection box just turn Internet Information Services on. This will automatically select almost all of the features that we need. We will do additional configuration steps later on

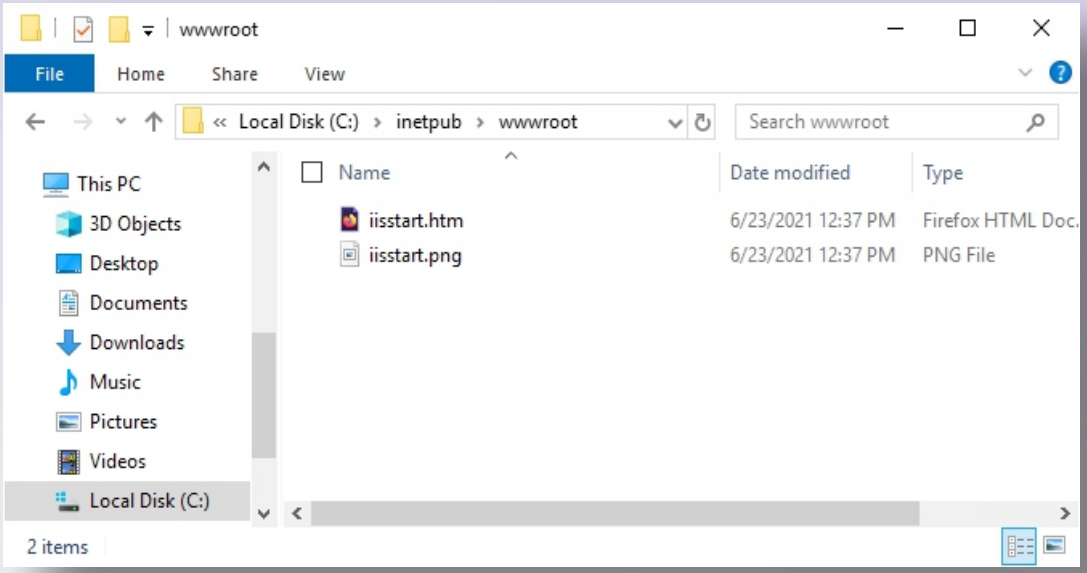


❺ Windows will now install IIS. We can check if the install was successful by navigating to localhost in our browser `http://localhost`



❻ We now have an IIS server running on our Windows-10 machine

The Internet Information Services server, or IIS server, can be used to serve several types of web site and web service content. The most apparent one is the html page that we just requested by visiting localhost. It's simply a file located in the IIS root folder.
`C:\inetpub\wwwroot`



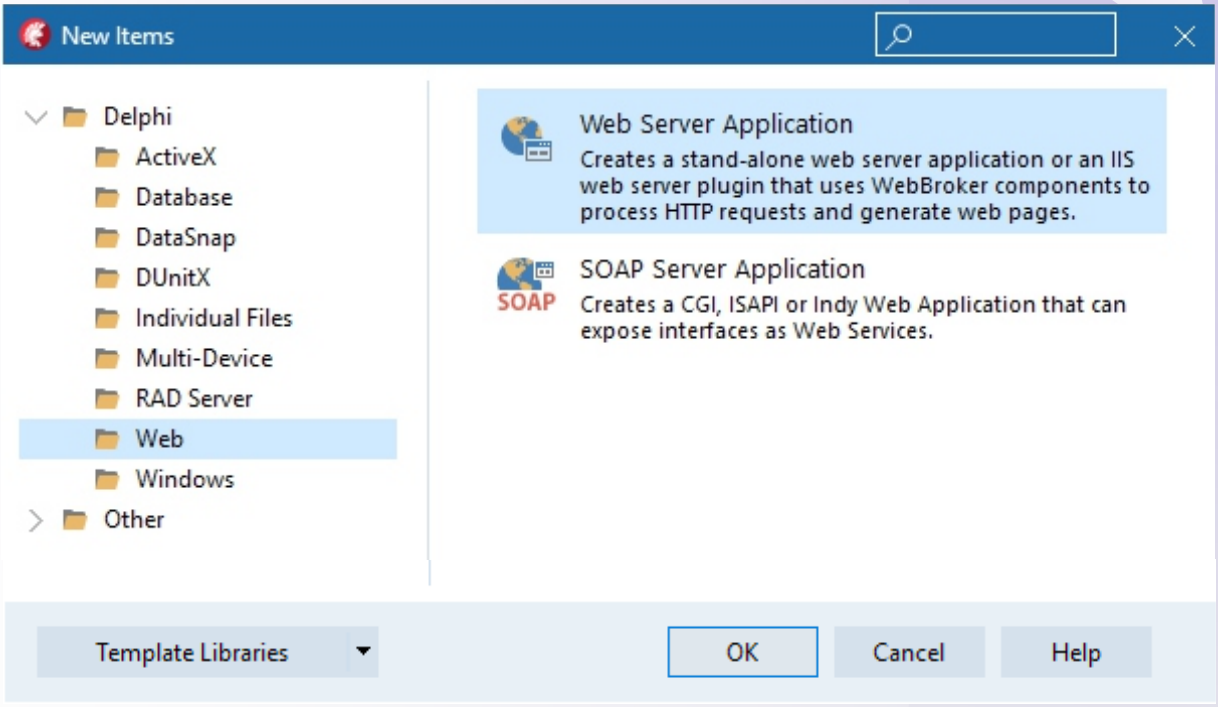
Any other htm file that you place at this location will be available in the IIS web server as a web page.

Please note that with installing IIS on your machine it also opened port 80 (HTTP) and port 443 (HTTPS) for external connections. At the end of the article we'll show you how to easily block or allow traffic to these ports in the Windows Firewall, as you may not want IIS available to everyone all the time, especially if you are on a public internet location, such as an airport.

One of the things that can be hosted by an IIS server is an ISAPI extension. ISAPI stands for Internet Server Application Programming Interface, and it allows us to create a dynamic link library (DLL) that adds to the functionality of the IIS server.

Our next step is to create an ISAPI dll in Delphi. The code from the web service we wrote in the previous article will be shared and embedded within this ISAPI dll.

❼ Create a new temporary Project Folder, eg \ISAPI
❽ In Delphi create a new Project with File | New - Other and under Delphi - Web choose Web Server Application

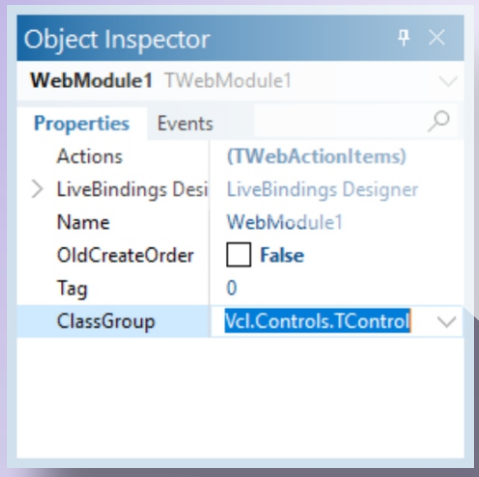


❾ In the wizard choose Windows and then ISAPI Dynamic Link Library

❿ Save the project as WebServiceISAPI in the temporary folder \ISAPI

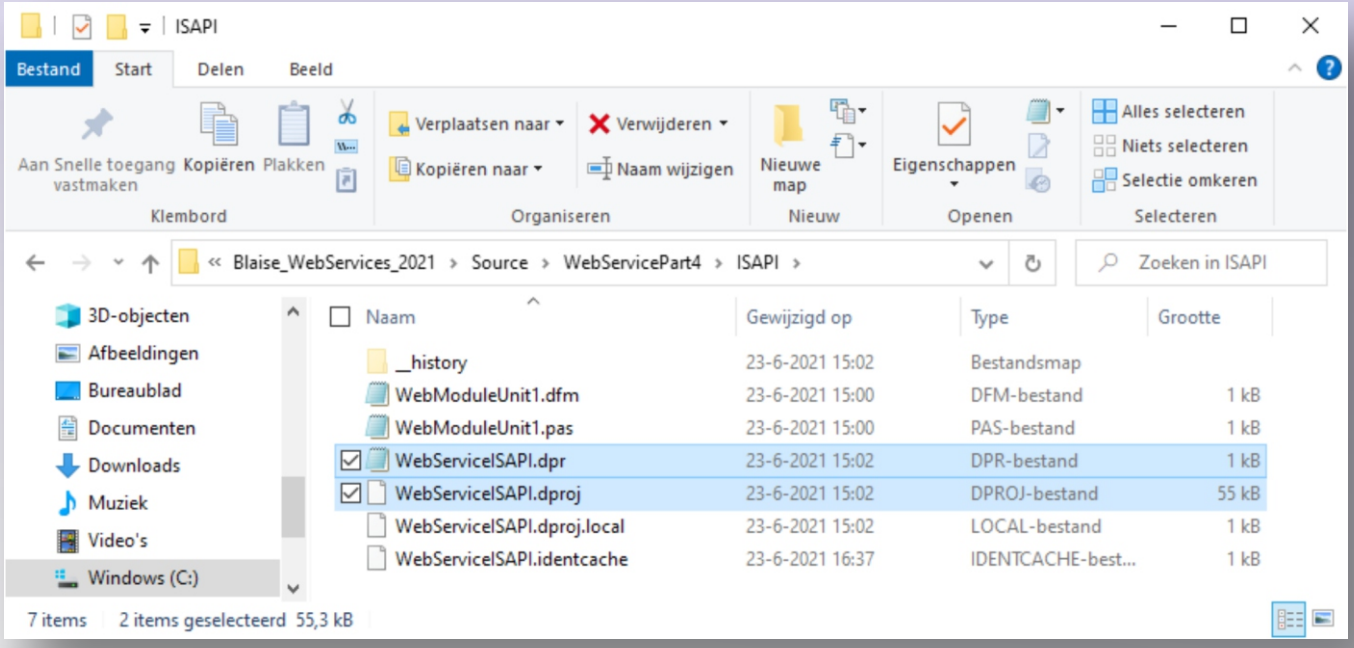
⓫ Now take a look at the project manager. Notice how the ISAPI project has the same structure as the previously created WebServiceServerGUI project. It is just a project file and a WebModuleUnit

⓬ There is a difference between the two WebModuleUnit files. The one from WebServiceServerGUI has the ClassGroup property set to the Vcl framework , while the one from the WebServiceISAPI has the ClassGroup property set to the framework neutral setting. We will change that after we merge the two projects

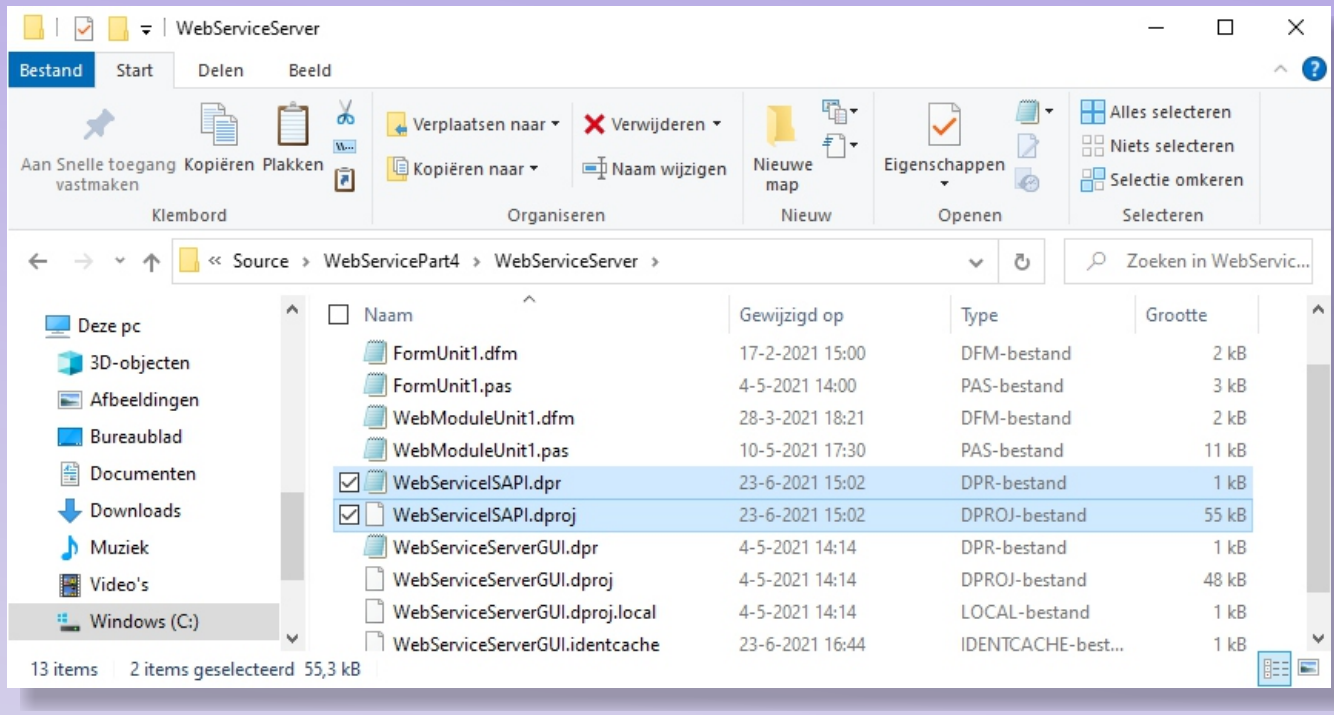


In order to merge the new ⓭ WebServiceISAPI with the existing WebServiceServerGUI we can simply copy the two WebServiceISAPI project files over to the WebServiceServerGUI directory. By doing that we let the new ISAPI project use the existing WebModuleUnit from the WebServiceServerGUI as they are in the same folder

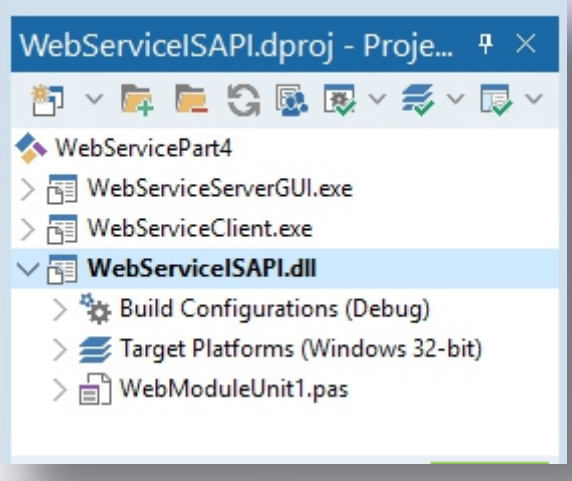
```
copy \ISAPI\WebServiceISAPI.dpr     \WebServiceServerGUI
copy \ISAPI\WebServiceISAPI.dproj   \WebServiceServerGUI
```

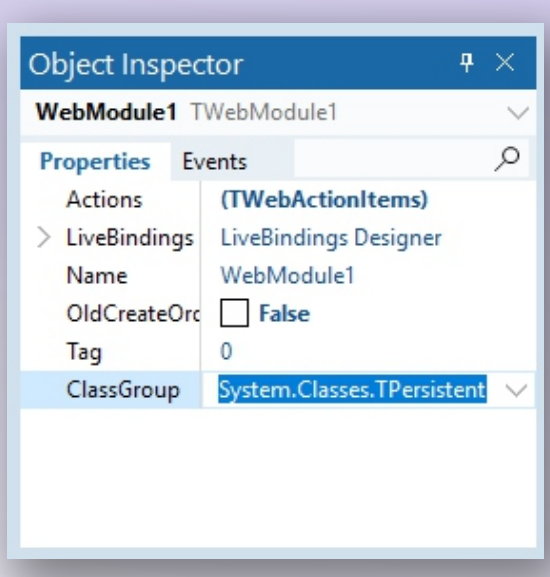


⓮ After copying these files over to the existing WebService project folder



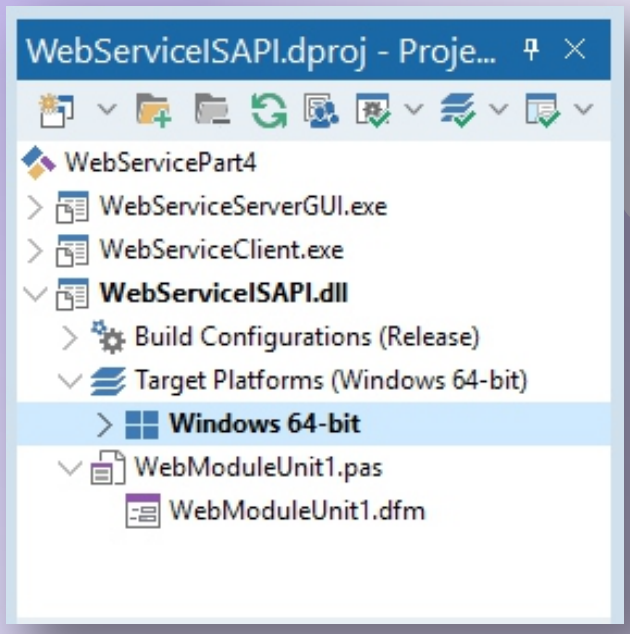
⓯ we can now add this new project to the Project Group with a right-click in the Project Manager and Add existing Project





⓰ Now open the `WebServiceISAPI` project from the Project Manager and open the `WebModuleUnit`. As this `WebModuleUnit` is now shared with the VCL `WebServiceServerGUI` we need to change the property `ClassGroup` to the framework neutral `System.Classes.TPersistent`

⓱ We also modify the target platform for this project to 64-bit, as ISAPI extensions in Windows-10 are by default 64-bit. Right click on the project in the Project manager and Add the Windows-64 bit platform



⓲ We do not need the 32-bit target platform for the ISAPI dll so you can remove it

⓳ Now build the WebServiceISAPI dll to test if merging the projects was successful

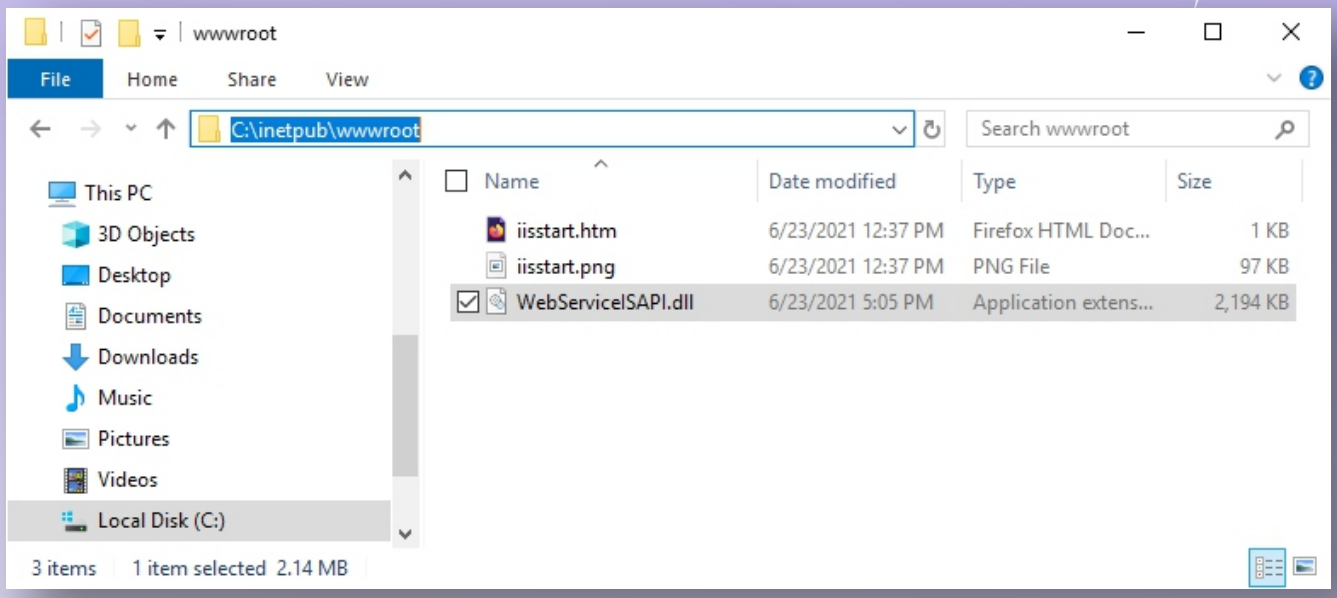
By merging the files from the VCL standalone application WebServiceServerGUI with the WebServiceISAPI dll project we can just develop and debug our web service with the VCL application, and then deploy it with a build of the WebServiceISAPI dll. It's much easier to debug a VCL application than debugging an ISAPI dll. Because the VCL version of our web service uses port 8080 and IIS by default uses port 80 we can even run them side by side.

Installing the ISAPI into IIS requires copying the dll file over to a directory that the IIS process can access as well as configuring IIS to load the ISAPI dll as an extension. For this article we will just use the DefaultAppPool and the wwwroot directory to run our ISAPI dll, because we are deploying to a development machine. In a production environment you may want to change the IIS configuration and create your own secured AppPool and website.

An Application Pool is a pool of resources within which an IIS application or website is run and which can be configured for the websites and applications that it governs. IIS comes pre-configured with the DefaultAppPool that services the default website at wwwroot.

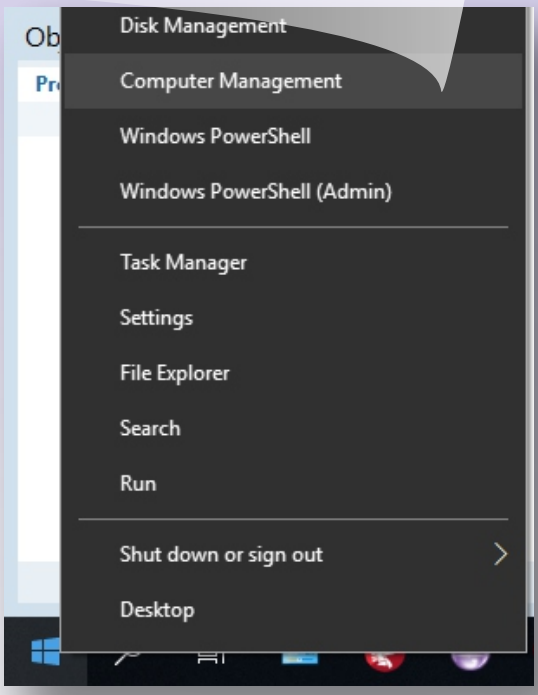
⓴ Copy the `WebServiceISAPI.dll` to the inetpub wwwroot folder. Windows may block copying from network locations to the protected inetpub folder, so you may need to copy it to an intermediate local folder `(C:\TEMP)` first

```
copy WebServiceISAPI.dll C:\inetpub\wwwroot
```

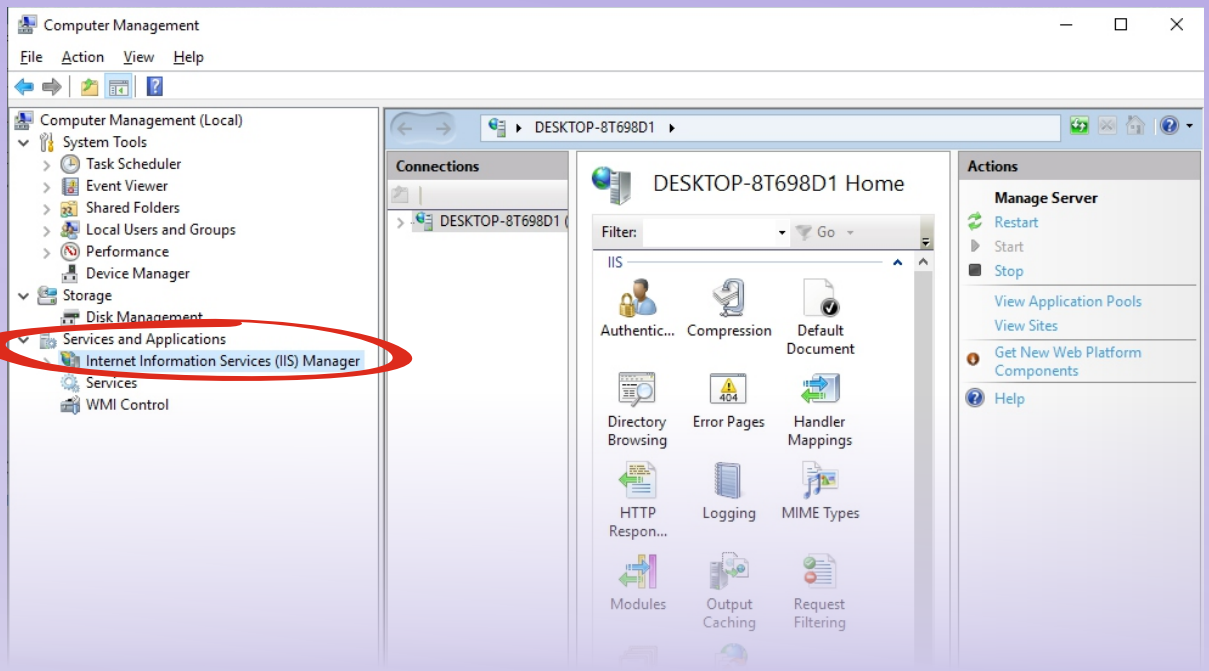


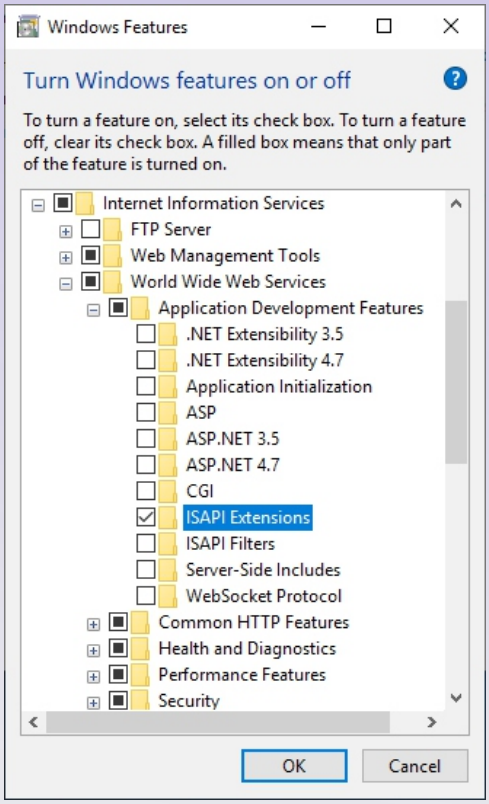
㉑ Next we need to configure IIS to allow execution of this ISAPI extension

㉒ Open Computer Management with a right-click on the start button



㉓ In Computer Management select the Internet Information Services manager under Services and Applications

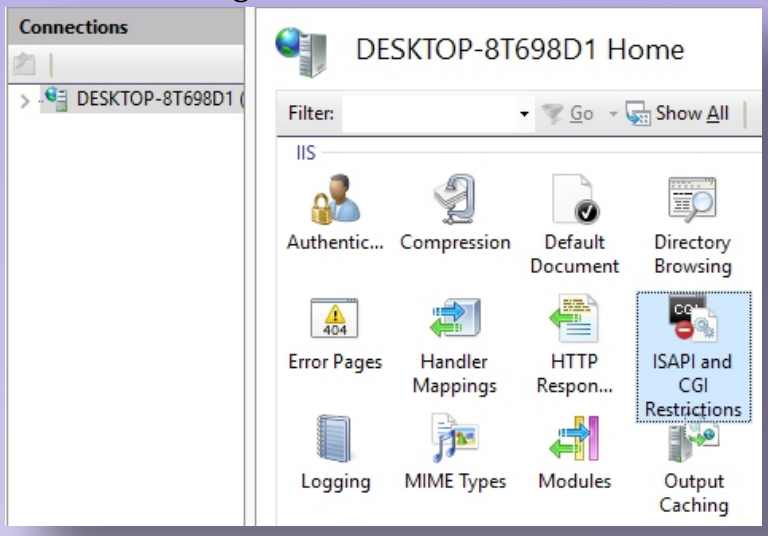




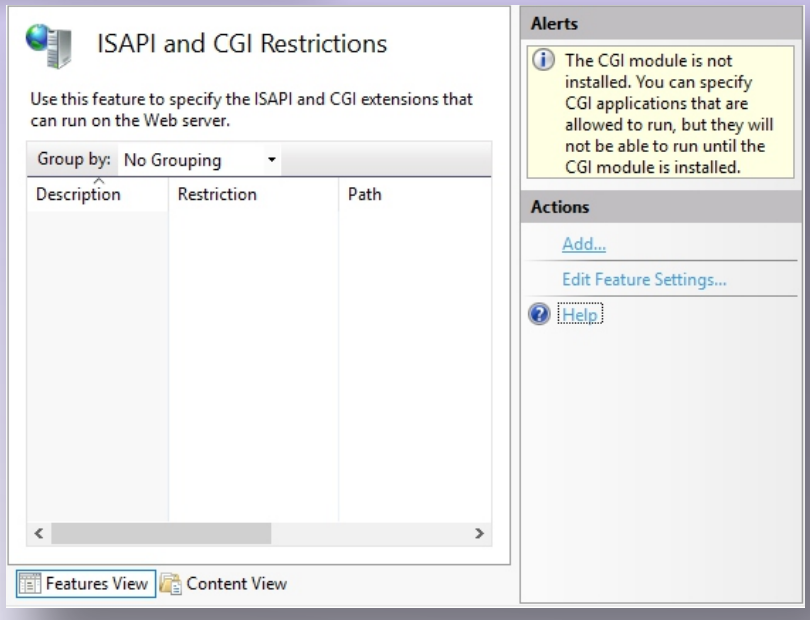
㉔ We need to use ISAPI/CGI Restrictions to configure our ISAPI, but as you can see the icon for that is not available. This is because in newer Windows and IIS versions ISAPI extensions are by default disabled and we need to enable it first.

㉕ Go back to the Control Panel as we did earlier and navigate to Turn Windows Features on and off. In the Internet Information Services branch we enable ISAPI extensions

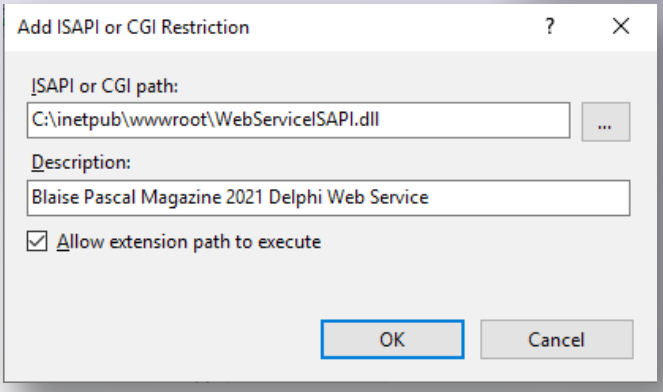
㉖ After enabling this we should see ISAPI/CGI Restrictions turn up in the IIS manager



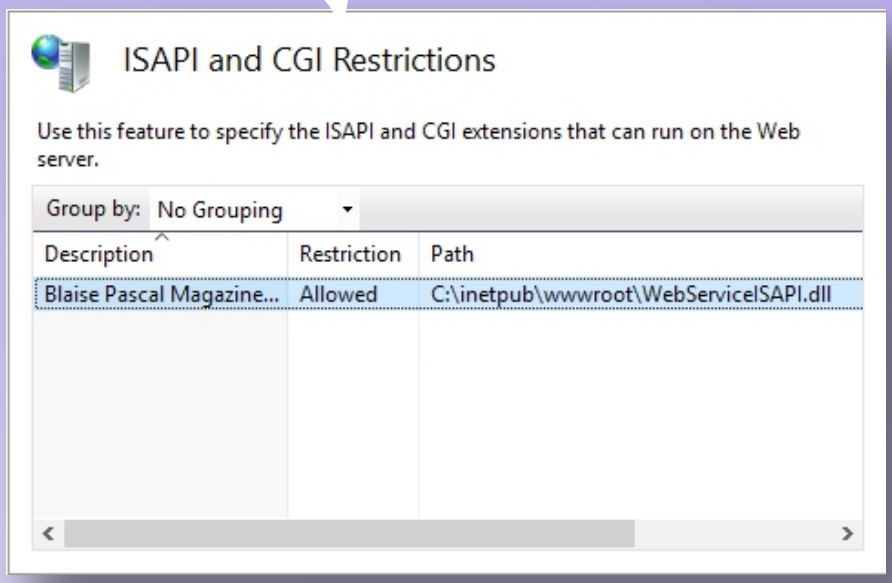
㉗ Open the feature and use the Add Action on the right to Add our ISAPI dll to IIS



㉘ We configure only this single ISAPI dll and allow it to be executed

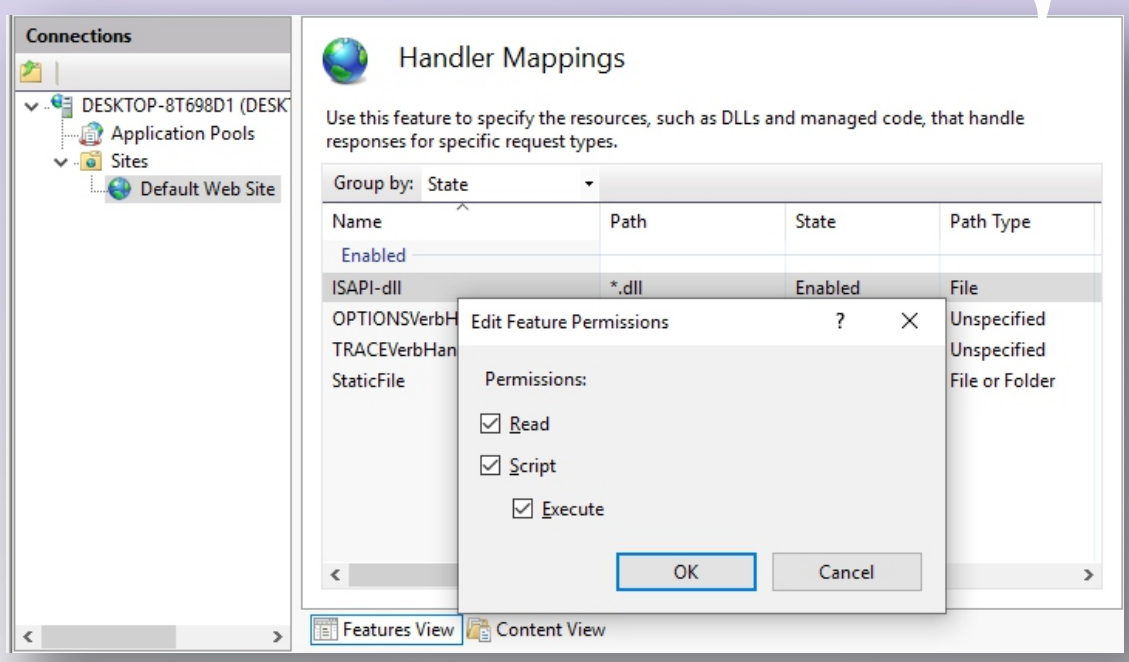


29 After adding the ISAPI dll you'll find it in the list

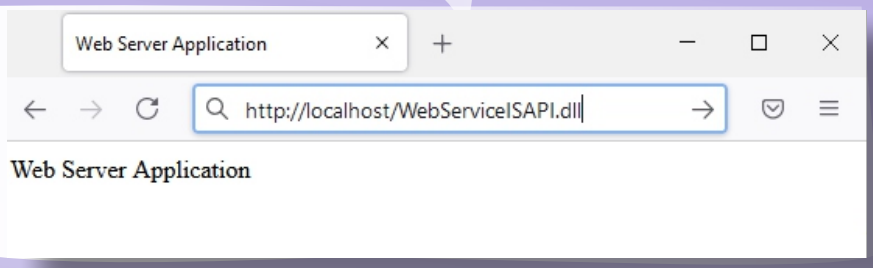


30 After adding the specific ISAPI to the allowed list we also need to configure handler mappings for wwwroot

31 Select the Default Web Site and open the Handler Mappings with the icon. Then right click on the ISAPI dll line and Edit Feature Permissions and enable execution

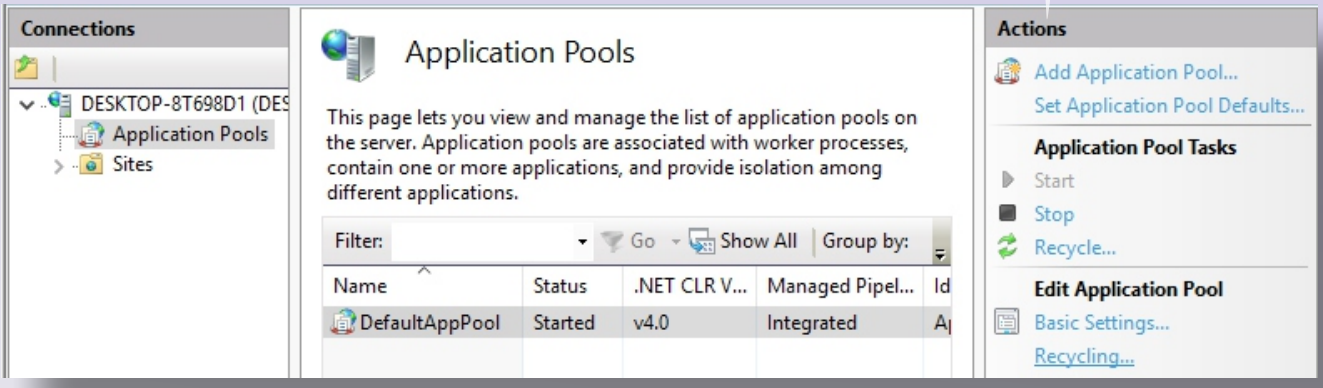


32. By pre-selecting the Default Web Site we can configure Handler Mappings just on this website. Handler mappings on server-level will still have ISAPI-dll as disabled
33. Test if this works by requesting the WebServiceISAPI.dll in a browser `http://localhost/WebServiceISAPI.dll`

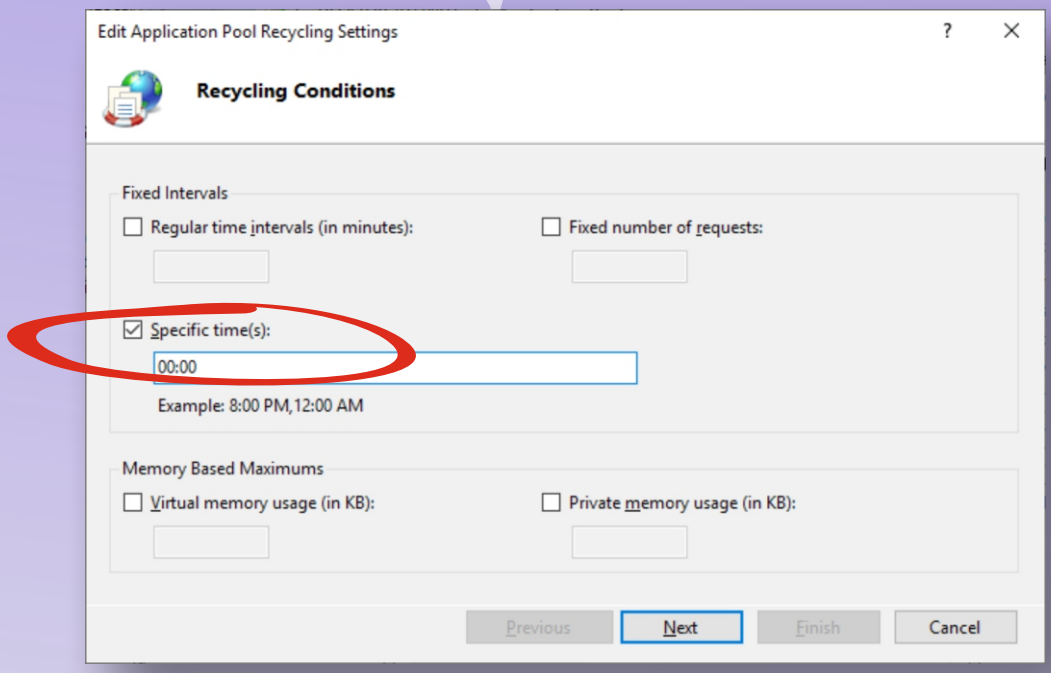


The web service is now up and running as an ISAPI extension. Because the web service uses transient in-memory storage and ISAPI extensions are unloaded by IIS when not in use, our web service currently suffers from forgetfulness. We need to reconfigure IIS to hold our web service in memory for a longer time period to be able to use it properly.

34. Select the Default Application Pool and open Recycling on the right-hand side under Edit Application Pool

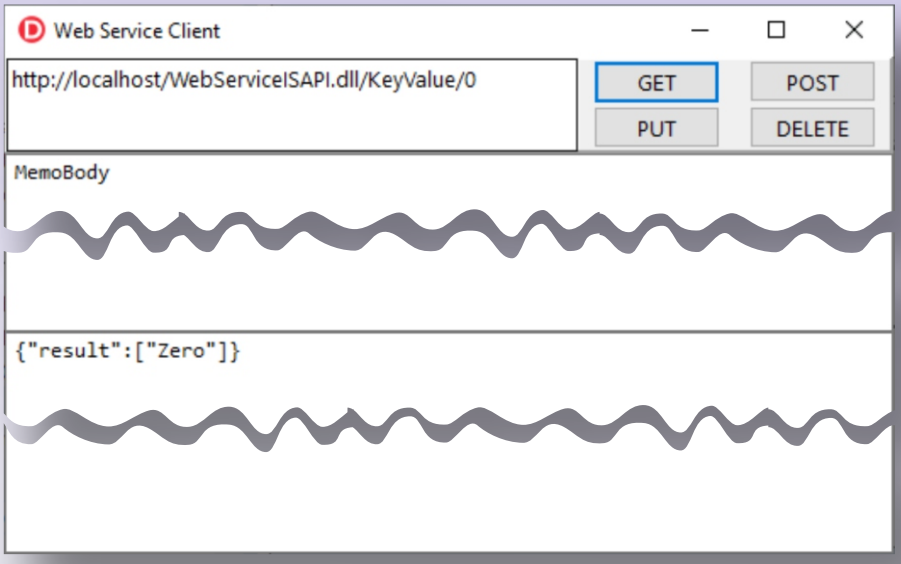


35 Modify the Recycle time and set it to 00:00 specific time



36 The IIS manager will translate this to the local time format, so it may end up as 12:00 AM depending on your international settings.
This is fine, its intended to Recycle at midnight, once every 24 hours

37 Test if it all works by reconfiguring the URL for the WebServiceClient application in the EditURL text `http://localhost/WebServiceISAPI.dll/KeyValue/0`



38 You can now POST and PUT new values with the Delphi web service client. If you close the Delphi client and re-open a new web browser and request the POSTed value it will still be there. If this does not work check the Recycling settings of the Default App Pool.

Our web service is now up and running on an IIS web server. The IIS web server can also be used to serve static HTML and JavaScript pages. In our previous article we added a /JavaScript Web Action Handler that returned some HTML and JavaScript code that in turn used the web service itself to return a number through a REST request on localhost\Number. It's a bit of a convoluted path. Now that we are running on a web server it makes much more sense to save this HTML with JavaScript page to a regular static html file in the default website and let the IIS web server handle the request.

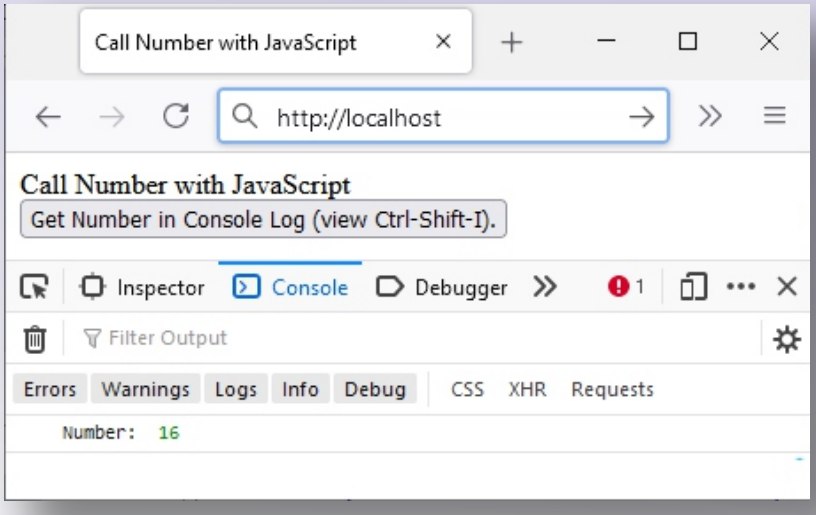
39 Copy the HTML and JavaScript code from the /JavaScript Web Action Handler to Notepad and change the embedded url to our new WebServiceISAPI. Don't forget to remove the additional single quotes or just use the below text

```
<html>
<head><title>Call Number with JavaScript</title></head>
<body>Call Number with JavaScript <button onclick="getNumber()">Get Number in
Console Log (view Ctrl-Shift-I).</button>
  <script type="text/javascript">
  function getNumber() {
   let url = 'http://localhost/WebServiceISAPI.dll/Number';
   fetch(url).then(resp=> resp.json().then(j=> console.log('\nNumber: ', j)));
  }
  </script>

</html>
```

Note that later on you may want to change the reference of localhost to your external IP or your DNS registered domain name, as localhost only makes sense from within your development machine. For now we need it to be localhost.

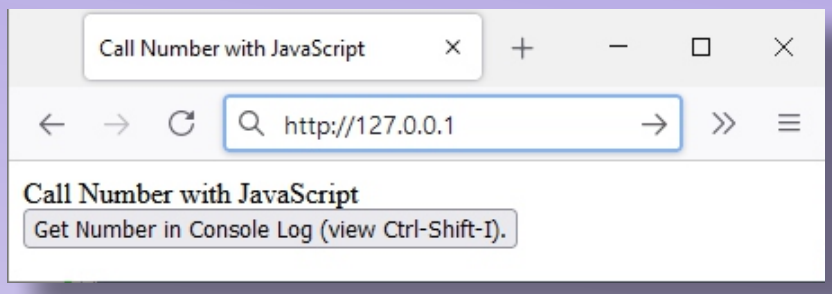
40 Save this file as default.htm in a folder \HTML and copy it to the IIS wwwroot folder
`C:\inetpub\wwwroot\default.htm`



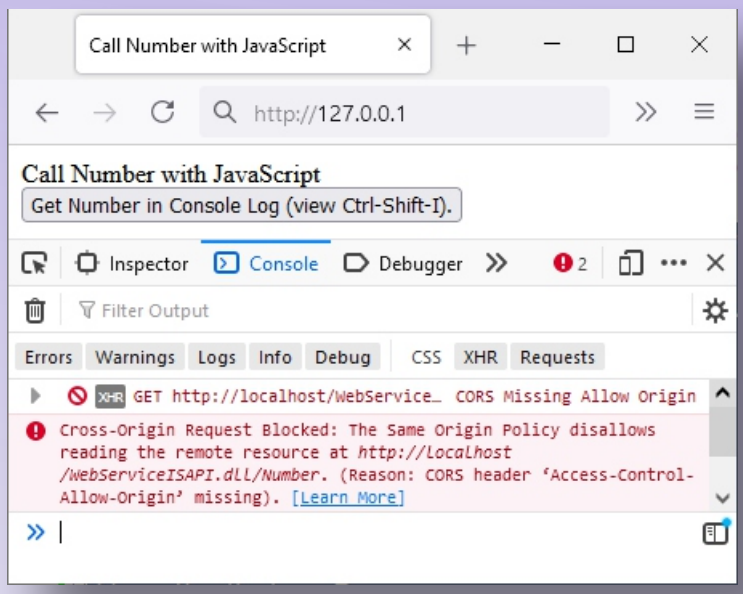
41 IIS will prefer loading default.htm instead of `iisstart.htm`, and the default HTML page will be displayed when visiting localhost

42 Test if it works by opening the localhost url in a browser from inside your development machine. Remember to open the Console Log in Firefox or Chrome/Edge with **Ctrl-Shift-I**

43. This all looks fine and the button works and requests a new Number from the new ISAPI web service
44. What if instead of using localhost we would use the url `http://127.0.0.1` ?



45. The page opens just fine.But if you now click on the Get Number button we get an error in the Console

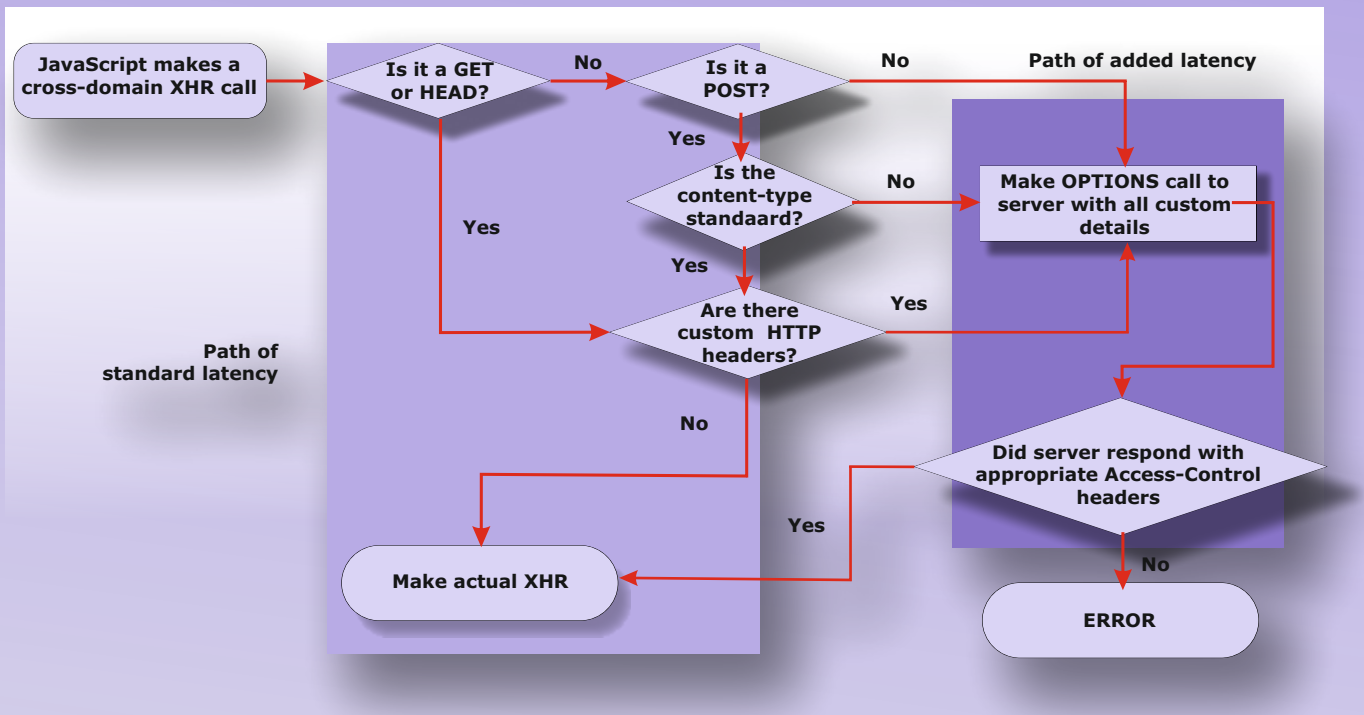


46. Let's investigate why this is happening
We visit the default.htm web page with the IP address 127.0.0.1. When we then click on the Get Number button, the JavaScript performs a XHR GET request from the web page at address 127.0.0.1 to a web service at localhost. We know that these are one and the same, but in fact, the localhost hostname and 127.0.0.1 are considered to be different by the browser, so the browser handles it as a cross-site event. The browser therefore asks the web service at localhost if this cross-site request should be allowed. Because the web service does not indicate that such a Cross-Origin Request is allowed, the request is blocked by the browser.

CORS Cross Origin Resource Sharing.
A web service can indicate access from other domains is allowed, by responding with 'Access-Control-Allow-Origin' in the HTTP header.

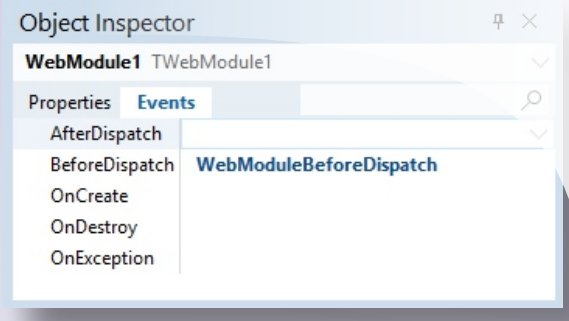
`https://en.wikipedia.org/wiki/Cross-origin_resource_sharing`

The following image from wikipedia on Cross Origin Resource Sharing illustrates this path.



So what we need to do is have our web service allow **CORS** by adding the appropriate `Access-Control*` headers.

47 Open the **WebModule** unit of the project and add a BeforeDispatch event-handler to the WebModule



48 Add **CORS** code to the **WebModule BeforeDispatch** event-handler, to indicate that calling our web service from other domains is OK.

```
procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject;
 Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  {Report back to the caller/browser that we allow Cross Origin Resource Sharing (CORS) for all calling domains}
 Response.SetCustomHeader('Access-Control-Allow-Origin','*');

  if Trim(Request.GetFieldByName(
   'Access-Control-Request-Headers')) <> '' then
  begin
    Response.SetCustomHeader( 'Access-Control-Allow-Headers',
       Request.GetFieldByName('Access-Control-Request-Headers'));
    Handled := True;
  end;
end;
```

The code above is the most permissive response you can have for a web service.

If someone initiates a CORS preflight request, all of the Access-Control-Request-Headers are allowed by simply copying them over to the Access-Control-Allow-Headers in the response. If a client asks if POST is supported, we indicate that it is, same for any of the other HTTP methods. We could also limit this a bit, to indicate that we support just a limited subset of HTTP methods by using Access-Control-Allow-Methods: POST, PUT, GET, DELETE.

In the same code snippet, by setting Access-Control-Allow-Origin to '*", we are allowing CORS calls from all origins. You could limit this to be less permissive by setting specific domain names in Access-Control-Allow-Origin.

If you start using your Delphi web service from other client platforms or browsers you may also run into caching issues. To prevent a web service consumer from caching previously retrieved values for the idempotent and cacheable HTTP GET command, you can set some headers in the web service to inform the web service consumer that it should not cache anything.

49 Add these Custom Headers to the code in the WebModuleBeforeDispatch handler to request no-caching

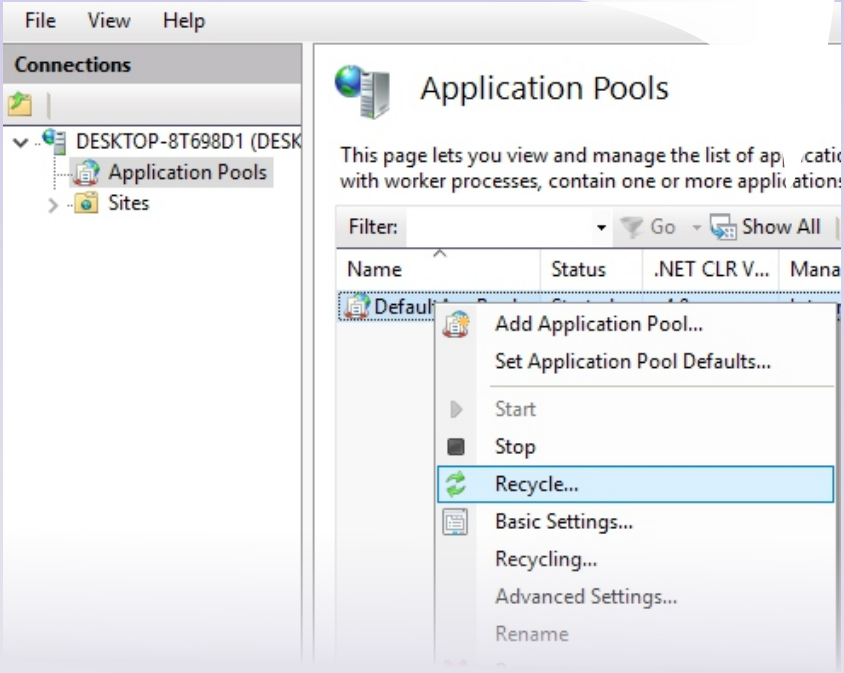
```
{ Set additional headers to ask client-side to not cache locally
  Cache-Control=no-cache, no-store, must-revalidate
  Pragma=no-cache
  Expires=0 }

Response.CustomHeaders.AddPair('Cache', 'no-cache, no-store, must-revalidate');
Response.CustomHeaders.AddPair('Pragma', 'no-cache');
Response.CustomHeaders.AddPair('Expires', '0');
```


These added headers ask other web service consumers to always retrieve new values for each GET request. This is not entirely foolproof though. You may run into caching proxy servers that choose to ignore these headers, in which case you would need to take additional steps. One of the tricks commonly used is to add a dummy parameter to the URL that the client then changes with each request. This dummy parameter can be ignored by the web services server, but any in-between caching proxy will see this as a completely new request and not serve the result from its cache. One example:

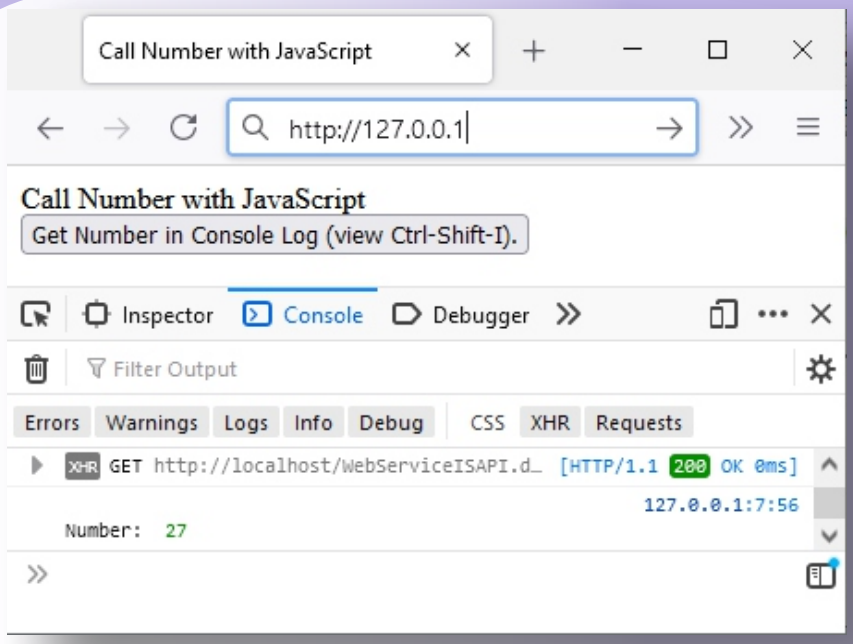
{This URL is different}
`http://localhost:8080/KeyValue?key=0&unique=random42`
{each time}
`http://localhost:8080/KeyValue?key=0&unique=random84`
{but gets the same value from the key value store}

50. We are now ready to deploy the next version of our `ISAPI dll`.
To update the dll, recompile the ISAPI library in Delphi and perform the following steps to replace the existing `ISAPI dll`
51. Open the IIS manager by using the **Search** field next to the Start button. Search for `"inetmgr"`
52. Open the Application Pools setting, select the Default Application Pool, right-click and choose Recycle.
By manually recycling the Application Pool of the ISAPI dll it will be unloaded from memory and you will be able to replace the dll file on disk



53 Then using the File Explorer remove the existing `WebServiceISAPI.dll` from wwwroot and copy the new version of this dll to the same location. If you are unable to replace the dll it may still be in use, you can release it by stopping and starting the IIS web server.

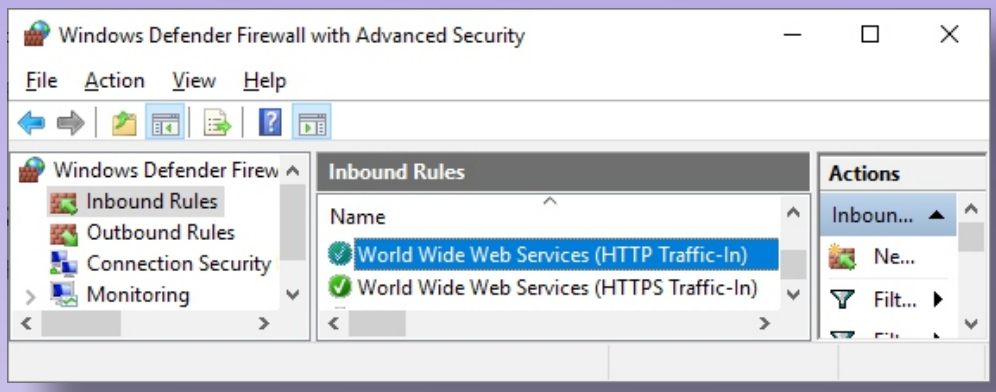
54 If we now try it again with `http://127.0.0.1` it works OK



With these modifications to the web service CORS preflight our web service will be accessible from external domains. Note that the localhost reference in the JavaScript snippet only makes sense from within the development machine. If you want to access the default.htm from outside of the development machine you need to change it to the external IP or hostname. For a production machine you'd change it to the domain name.

The web service is now running under IIS and the coding is done, but there are some additional tips and hints that I'd like to share.

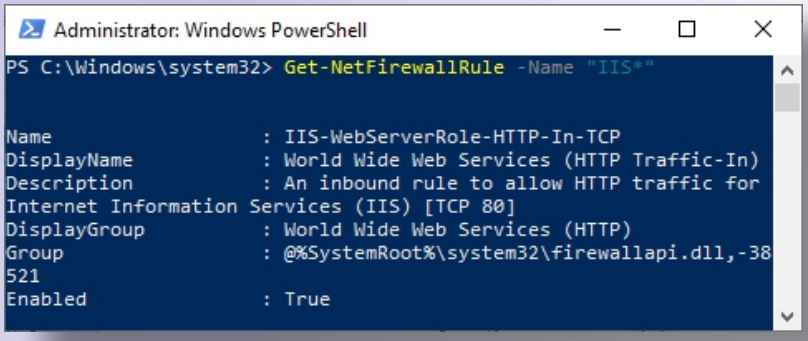
When we installed IIS on the development machine it also opened up port 80 (HTTP) and port 443 (HTTPS) for all types of networks `(Private, Public, Domain)`. When you visit a location with public internet, such as an airport, you may not want to have these IIS ports open. Of course you can easily change these firewall rules, and close these ports, using the Advanced Firewall configuration.



But this does require some mouse clicks and there is an easier way of doing this using PowerShell. Right-click on the start menu and start Windows PowerShell (Admin).

55 Get a list of the two firewall rules for IIS with this command

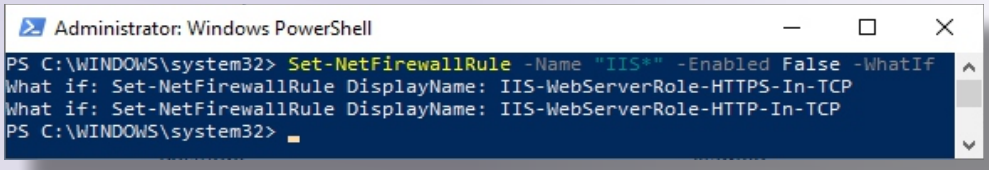
```
Get-NetFirewallRule -Name "IIS*"
```



56 Verify that you see just two rules, for port 80 and 443, both for IIS

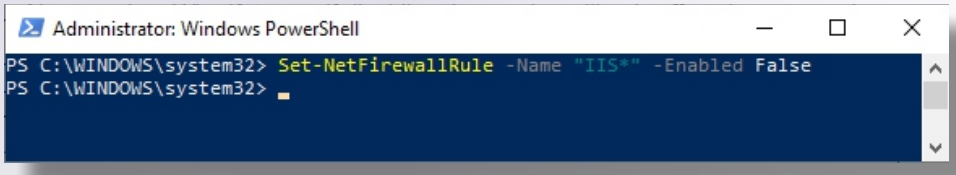
57 Next we do a WhatIf, to see if disabling these rules will only affect these two rules

```
Set-NetFirewallRule -Name "IIS*" -Enabled False -WhatIf
```

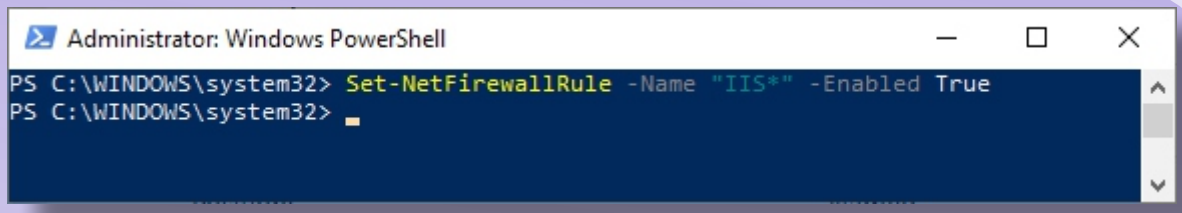


58 If that looks good then execute the actual disable command

```
Set-NetFirewallRule -Name "IIS*" -Enabled False
```



59. Now test if IIS can still be reached on your machine from another machine using your external IP address. It should now be blocked. Note that your localhost will remain operational, as it's only blocking external connections.
60. Unblocking is easy as well, just use the -Enabled True command
`Set-NetFirewallRule -Name "IIS*" -Enabled True`



61. And IIS can again be reached from other machines.

While you are actively developing and testing with Delphi on IIS on your development machine or laptop you can easily enable the firewall rule for full access. In all other cases you can disable it and have IIS safely hidden behind the firewall.

In this deployment we chose to host our ISAPI dll in the Default Website and the Default Application Pool, which is fine for a development machine. However for deployment on a production environment you will probably create a new website and a new Application Pool for each ISAPI dll. This makes it easier to manage each ISAPI extension as well as more secure. Additionally you can shorten the URL to the WebServiceISAPI.dll with an URL Rewrite, which makes it both easier to access the web service as well as obscures the actual ISAPI dll filename. For production you will also need to harden your IIS installation, making it even more secure.

In our next article we'll be deploying our web service to Apache on Linux. Stay tuned!

Source code for this article can be downloaded here:
`https://www.blaisepascalmagazine.eu/your-downloads/`