

kbmMW and AJAX

(or how to make a cool webservice using kbmMW)

AJAX is the current new buzzword of the IT business. What is AJAX apart from being a namebrother to SOAP (the type you do dishwashing with)?

AJAX means Asynchronous Javascript and XML. I suspect that the acronym was invented before the meaning of it.

For a long time now, we have gotten used to web browsers accessing web servers that provides HTML files, graphic files and browser script files (like Javascript) to the browser. The operation was usually that the complete page was requested and loaded from the webservice.

AJAX doesn't really provide any new technologies compared to that, but it use the elements in a slightly different way.

To call something AJAX based, it basically means that you upload some html and Javascript to the browser as the result of the browser requesting it (exactly as usual from a web browser), but from there on, things differ somewhat. The Javascript provided for the webbrowser now takes over the operations of the web site, and instead of POST'ing or GET'ing completely new webpages from the webservice, the client just requests specific pieces of data that it needs, and lets the Javascript running in the client update the document(s) in the browser dynamically.

With the right AJAX libraries (essentially Javascript based applets), its possible to design a thin webbased application that very much looks like a normal standalone application.

When buttons are clicked, the complete page does not automatically redraw completely, but instead only the bits needed are updated.

All this require quite a lot of Javascript code though, but fortunately there exists lots of Javascript libraries making the job easier.

The XML part of the AJAX acronym comes in due to the fact that one of the preferred ways to provide data to the webservice about the AJAX call, and to receive data from the webservice, is to use XML. However this is not mandatory, and many AJAX libraries prefer to use JSON which essentially is the way that Javascript serializes its objects.

AJAX client side

Within Javascript exists a method that can be called to make a call to a webserver from the script. Unfortunately the standard AJAX is not much standard when looking at browsers... there is the Microsoft way and there is the way most others are using.

The following code shows how to instantiate the relevant object within Javascript that makes an asynchronous callback to a webserver possible:

```
function createRequestObject() {
    var ro;
    var browser = navigator.appName;

    If (browser == "Microsoft Internet Explorer") {
        ro = new ActiveXObject("Microsoft.XMLHTTP");
    } else {
        ro = new XMLHttpRequest();
    }
    return ro;
}
```

createRequestObject creates an instance of the object that is used for calling the webserver. It takes the browser type into account as there are multiple ways to instantiate such object.

```
var http = createRequestObject();
```

Instantiate an instance of the http object that we will use to call the webserver.

```
function sndReq(action) {
    http.open('POST', 'WebForm2.aspx?action='+action);
    http.onreadystatechange = handleResponse;
    http.send(null);
}

function handleResponse() {
    if (http.readyState == 4) {
        var response = http.responseText;
        alert(response);
        document.getElementById('MessBox').value = response;
    }
}
```

sndReq is the method that we can call to send the request to the webserver. Notice that the call is non blocking. Thus we arent waiting for the result. Instead the function handleResponse is executed the moment the result is available. That's where the asynchronous part of AJAX comes into play.

This small sample will just call a WebForm2.asp page on a webserver with the query string action=someaction, and will display the response, when its ready, as an alert.

That's the basic principle of AJAX. Pretty easy huh ☺

The AJAX server

An AJAX server is a web server! Typically with the ability to execute server side code or scripts to provide dynamic logic for the client calls.

This is where kbmMW enters the scene. kbmMW v. 2.70 Pro/Ent has gained the ability to mimic a webserver, while continuing to work like a normal application server the way you are used to.

A kbmMW based AJAX server simply require you to set the transport streamformat to AJAX. Without doing anything more, webbrowsers can now make AJAX requests directly to the functionality provided by the application server.

The kbmMW AJAX streamformatter supports two syntaxes for accessing the kbmMW application server:

- 1) Calling it with a body of XML, resulting in XML based output
- 2) Calling it with anything else

Using XML

If the webbrowser wants to call the application server using the XML format, the browser must call using this syntax:

<http://yourserver:optionalport?XMLREQUEST>

and then provide a kbmMW compatible XML request as the body. A kbmMW compatible response will be returned to the client. Please read the whitepaper 'kbmMW XML transport format' for the XML syntax to be followed. (notice the XMLREQUEST word, which must be specified as shown).

Hence any service that the application server allow the client to call, can be called this way very easily as the XML format allow you to specify servicename, version, function name, arguments etc.

Not using XML

The other option is to call without the XMLREQUEST argument.

Like this:

<http://yourserver:optionalport/someoptionalstuff/~servicename.serviceversion/moreURL?query>

This way you can call a specific service. The function name will then be one of the following: **GET, PUT, HEAD, POST, DELETE, TRACE, CONNECT** or whatever the webclient provides as the HTTP operation.

The '*moreURL*' part is automatically provided as Args[0], the provided Mimetype as Args[1], and the query string as Args[2].

The (dot)serviceversion part is optional.

In case the hazeltine (~) syntax is not specified, a service named HTTPSERVICE will be called.

Eg:

<http://yourserver:optionalport/moreURL?query>

Summary:

Functionname = HTTP operation

Args[0] = moreURL

Args[1] = mimetype

Args[2] = query string

Name of service called is HTTPSERVICE unless the hazeltine syntax is used.

This way its possible to create multiple virtual web servers within the same application server, by simply referereng to different services (and versions).

The function of the service that is being called should then return some value as the Result of that function.

The Result can be a variant or a one dimensional, non nested, array of variants. In case an array of variants is returned, each element of the array will be provided sequentially to the browser as a combined result.

This method is preferred when the AJAX client side library wants to communicate using JSON, or when you simply want to define your own type of data to return.

Eg.

```
Result:='<html><body>This is the result</body></html>';
```

and

```
Result:=VarArrayCreate([0,4],varString);  
Result[0]:='<html>';  
Result[1]:='<body>';  
Result[2]:='This is the result';  
Result[3]:='</body>';  
Result[4]:='</html>';
```

will both result in the same output.

JSON?

JSON is an alternative to XML to transmit Javascript data. On the client side, a wellformatted JSON string is automatically converted to Javascript objects and arrays.

The server side do not natively understand JSON, but it can easily learn that via one of several 3rdparty libraries.

I currently prefer LKJSON which can be downloaded here:

<http://sourceforge.net/projects/lkjson>

Read more about JSON at www.json.org



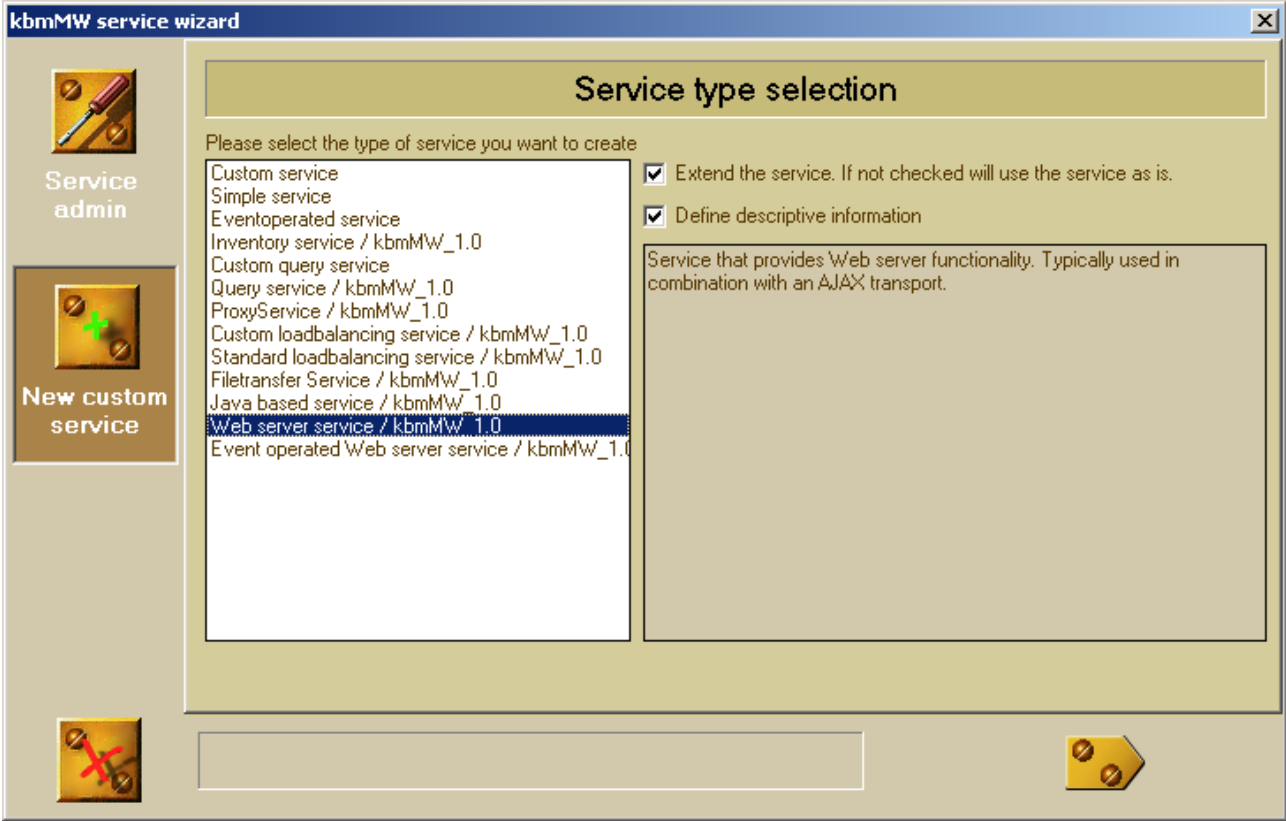
kbmMW as a WebServer?

Now you could just build your own custom service and call those from the web browser. However kbmMW already comes bundled with a couple of custom services you can use for creating a web server service very easily, and a large library of web related utility routines.



TkbmMWCUSTOMHTTPService

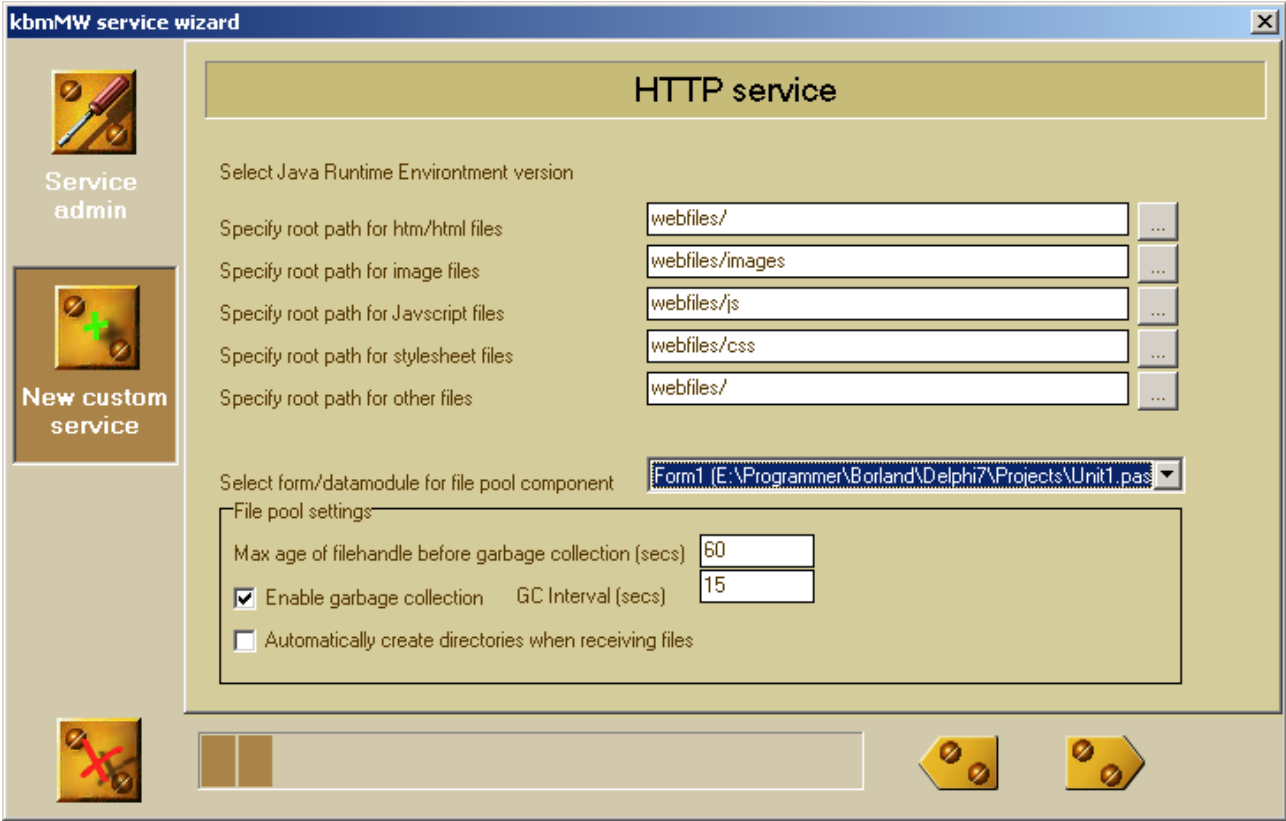
You can base your own HTTP service on this service by using the service wizard to create it



Select 'Web server service' and click the right arrow button.



This shows the special HTTP service wizard page. Within it you can define the settings for where web files are located and where to place a file pool component. The file pool is the shared access point and pool of file handles where the handles are provided for the files that the client requests.

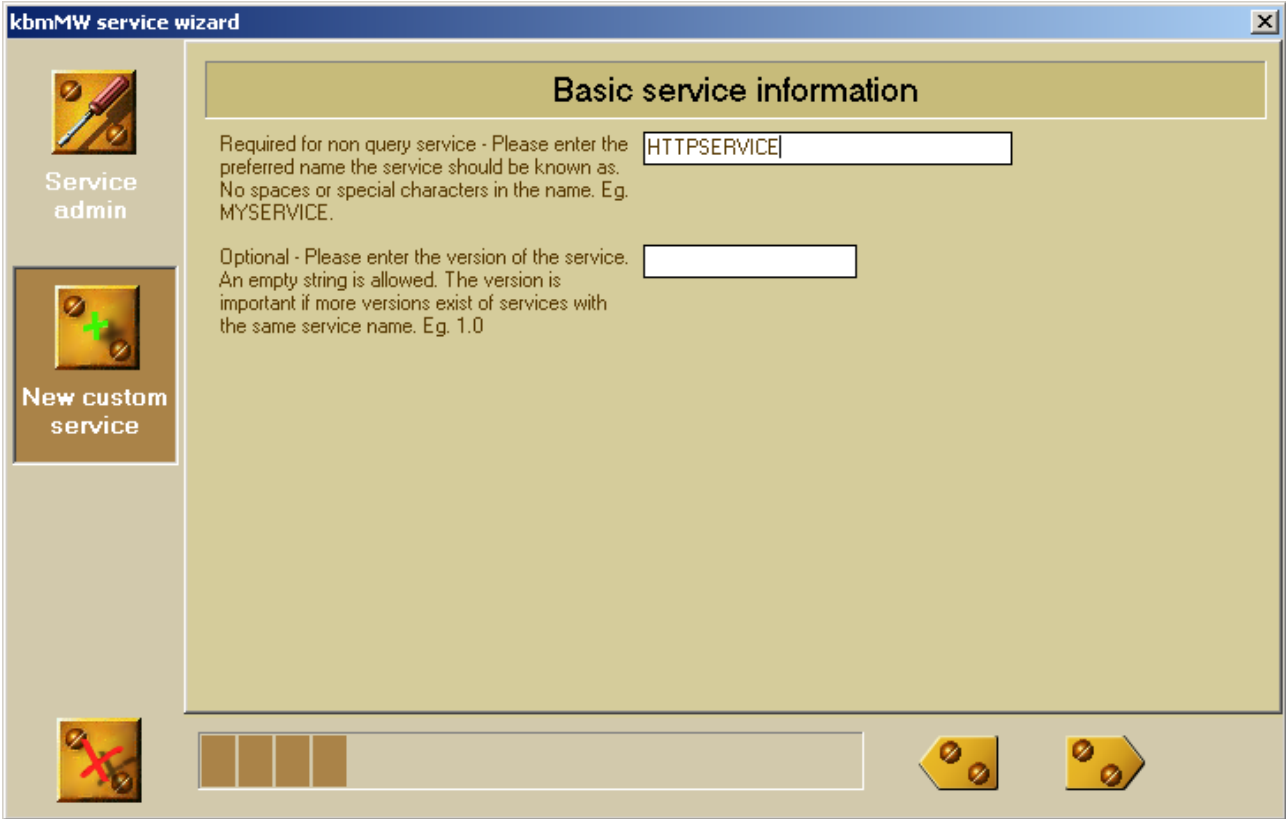


You would typically put the filepool on a shared datamodule/form. You can at any time change these settings on the file pool instance, and on the returned service definition when registering the HTTP service to the kbmMW application server. Cast the service definition to `TkbmMWHTTPServiceDefinition` to access the special HTTP properties.

Click the right arrow button.



Next step is like with any other services, to define the service name. Remember that if you want to use the non XML, non hazeltine access method (described earlier), you need to name this service HTTPSERVICE.



Go through the remaining steps in the wizard and generate the code.

Now register the service the normal way with the kbmMWServer component, and copy/paste the servicedefinition description from the comments in the top of the generated service to where you register the service.

Now you essentially have a complete webserver available.

It already contains functionality to respond to browser GET operations, which means that its able to display static web sites.

You could have chosen to base your service on the TkbmMWEventHTTPService by selecting 'Event operated web server service' instead. The difference is that the even service provides you with events for the most used browser operations (GET, PUT, HEAD, POST, CONNECT, DELETE), while the custom web server service require you to override some methods.

The service already knows about how to translate file types to mime types etc.
The following mimetypes are already predefined for you in the web service:

Graphics

GIF=image/gif
JPG=image/jpeg
PNG=image/png
JPE=image/jpeg
BMP=image/bmp
JPEG=image/jpeg
TIFF=image/tiff
TIF=image/tiff
XBM=image/x-xbitmap
TIFF=image/tiff

Text

HTM=text/html
HTML=text/html
RTF=text/richtext
TXT=text/plain
CSS=text/css
XML=text/xml

Applications

JS=application/x-javascript
PDF=application/pdf
BIN=application/octet-stream
EXE=application/octet-stream

DOC=application/msword
DOT=application/msword
AI=application/postscript
EPS=application/postscript
PS=application/postscript
GTAR=application/x-gtar
GZ=application/x-gzip
CLASS=application/octet-stream
SER=application/x-java-serialized-object
JAR=application/x-java-archive
TAR=application/x-tar
ZIP=application/zip
XLA=application/vnd.ms-excel
XLC=application/vnd.ms-excel
XLM=application/vnd.ms-excel
XLS=application/vnd.ms-excel
XLT=application/vnd.ms-excel
XLW=application/vnd.ms-excel
MSG=application/vnd.ms-outlook
POT=application/vnd.ms-powerpoint
PPS=application/vnd.ms-powerpoint
PPT=application/vnd.ms-powerpoint
MPP=application/vnd.ms-project
WCM=application/vnd.ms-works

WDB=application/vnd.ms-works
WKS=application/vnd.ms-works
WPS=application/vnd.ms-works
HLP=application/winhelp
MDB=application/x-msaccess
SWF=application/x-shockwave-flash

Audio

UA=audio/basic
WAV=audio/x-wav
MID=audio/x-midi
AIF=audio/x-aiff
AIFC=audio/x-aiff
AIFF=audio/x-aiff

Video

MPG=video/mpeg
MP2=video/mpeg
MPEG=video/mpeg
MPE=video/mpeg
QT=video/quicktime
MOV=video/quicktime
AVI=video/x-msvideo
MOVIE=video/x-sgi-movie

If you want to alter the mimetypes or add new, simply update the MimeTypes property of the service. A good place to do that is in the ConstructService method. The MimeTypes string list is used for determining the mimetype for specific files requested by the client.

Dynamic content?

You can override any of the following protected methods to provide custom functionality:

```
function PerformGET(ClientIdent:TkbmMWClientIdentity; const Args:array of Variant):Variant;
function PerformHEAD(ClientIdent:TkbmMWClientIdentity; const Args:array of Variant):Variant;
function PerformPOST(ClientIdent:TkbmMWClientIdentity; const Args:array of Variant):Variant;
function PerformPUT(ClientIdent:TkbmMWClientIdentity; const Args:array of Variant):Variant;
function PerformDELETE(ClientIdent:TkbmMWClientIdentity; const Args:array of Variant):Variant;
function PerformCONNECT(ClientIdent:TkbmMWClientIdentity; const Args:array of Variant):Variant;
```

As previously mentioned, the Args array will contain the following:

```
Args[0]=URL
Args[1]=Mimetype provided by the client
Args[2]=Querystring
```

The body (if any) of the request is in the **RequestStream** of the service.

The following service methods are available:

```
function GetMimeType(const APath:string):string;
```

Returns the mimetype for a given local file.

```
function FileCategory(const APath:string):TkbmMWHTTPFileCategory;
```

Returns the category of typical web files.

```
function HTTPResponseFromFile(const APath:string; var AMimeType:string):string;
```

Returns the content of the given file as a string which directly can be used as a result from one of the operation methods. The mimetype for the file is returned in AMimeType.

```
function SetResponseMimeType(const AMimeType:string):string;
```

Sets the mimetype of the response.

The HTTP headers of the request are available via the transport stream helper object that must be cast to `TkbmMWHTTPTransportStreamHelper`. Eg.

```
function TMyHTTPService.PerformPOST(ClientIdent:TkbmMWClientIdentity;
                                     const Args:array of Variant):Variant;
var
  helper:TkbmMWHTTPTransportStreamHelper;
begin
  helper:=TkbmMWHTTPTransportStreamHelper(RequestTransportStream.Helper);
  Result:='<html><body>The HTTP headers contains:<br>'+helper.Headers.Text+
    '</body></html>';
end;
```

The unit `kbmMWHTTPUtils` contains lots of additional functionality to assist in accessing, and generating HTML related data.

For example unpack the provided querystring. Eg.

```
var
  qv:TkbmMWQueryValues;
begin
  qv:=TkbmMWQueryValues.Create;
  try
    qv.AsString:=Args[2]; // Args[2] contains the query string.

    // Now access the name/pair values as you see fit via Value and ValueByName.
    Result:='<html><body>The result is:'+qv.ValueByName['edit1']+'
      '</body></html>';
  finally
    qv.Free;
  end;
end;
```



Build template based results via the template routines. Eg.

```
function kbmMWProcessTemplate(const ATemplate:string;  
    AVariables:TStrings;  
    const AVariableType:TkbmMWTemplateVariableType):string;
```

The template is a string which you can load from a file or from anywhere you see fit. The template can contain variable references. The format of those variable references depends on the AVariableType setting:

```
TkbmMWTemplateVariableType = (mwtvtStandard,mwtvtHTML);
```

mwtvtStandard indicates that variable references have the following format:
[!--variablename--!]

mwtvtHTML indicates that variable references have the following format:
<!-- variablename -->
(includes a space before and after the variable name)

The variablenames must refer to a name in the AVariables stringlist.

The returned result is the processed template.

Sample

For an advanced sample, check the AJAX_IM demo bundled with the kbmMW installation (Pro and Enterprise only).

It was originally an AJAX application that communicated with via a web browser with a PHP backend hooked up to a MySQL database.

The serverside parts were however ported to use kbmMW as the webserver, and Borland Database Engine as the database via a kbmMW cross adapter.

Links

<http://www.ajaxprojects.com/>

Contains amounts of AJAX utilities and documentation.

<http://www.json.org>

The place to read about JSON

This concludes the whitepaper about AJAX and kbmMW.

Kim Madsen
Components4Developers