4 Nations Locs TAB CC 72080 , Märklin 221 107-6 , SBB Re 4/4 II 11161, ACME E 444 014

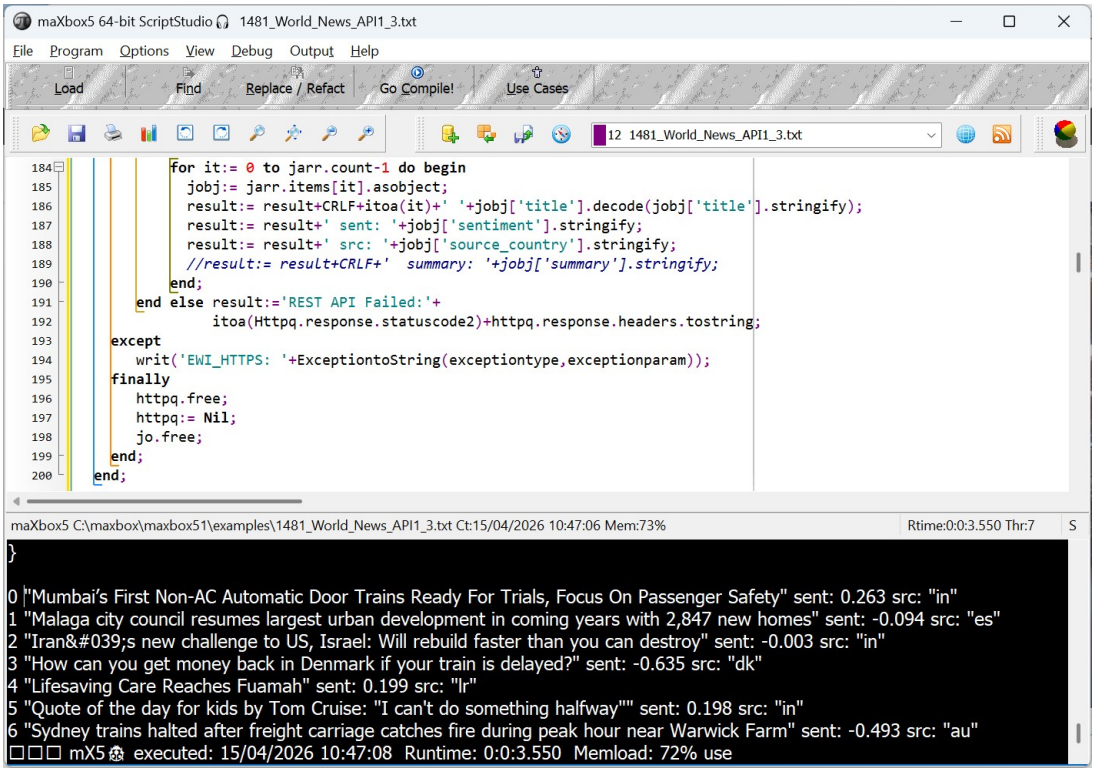
Best Rest Client

HttpComponent is a small Delphi wrapper around the WinInet library, written in Delphi 2010 and used in maXbox5. It turns the low-level Windows HTTP API into a component-style interface that is easier to drop into scripts and applications.github+1

maxkleiner/HttpComponent: Delphi component wrapper for WinInet library

What it does

The wrapper provides a simple request object with methods such as GET and POST, plus a response object for reading status codes and returned content. It also supports multipart uploads and URL-encoded form posts, which makes it practical for REST-style APIs and file transfer tasks.github+1



world news API with HTTPRequestC

Why it is useful in maXbox5

For maXbox5, the component fits well because it keeps HTTP code compact and readable inside a scripting environment built on Delphi. Instead of dealing directly with the raw WinInet calls, you can create a request component, add headers, send the request, and inspect the response in a few lines.blogs.embarcadero+2

Example use

A typical GET request is as short as calling `Get('https://httpbin.org/get')` and then reading `Response.ContentAsString`. For POST requests, the wrapper supports plain text bodies and multipart form bodies, including file attachments through `AddFromFile`.github+1

```
MultiPart FromData Stream
function TestHTTPClassComponentAPIDetection2(AURL, askstrea
var HttpReq1: THttpRequestC;
    Body: TMultipartFormBody;
    Body2: TUrlEncodedFormBody;
begin
  Body:= TMultipartFormBody.Create;
  Body.ReleaseAfterSend:= True;
  //Body.Add('code','2','application/octet-stream');
```

```
  Body.AddFromFile('image', exepath+'randimage01.jpg');
  HttpReq1:= THttpRequestC.create(self);
  HttpReq1.headers.add('X-Api-Key:'+AAPIKEY);
  HttpReq1.headers.add('Accept:application/json');
  try
    if HttpReq1.Post1Multipart(AURL, body) then
       writeln(HttpReq1.Response.ContentAsString)
    else Writeln('APIError '+inttostr(HttpReq1.Response.Sta
  finally
    writeln('Status3: '+gethttpcod(HttpReq1.Response.status
    HttpReq1.Free;
    sleep(200)
    // if assigned(body) then body.free;
  end;

In this code snippet:

    {
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "application/json",
    "Accept-Encoding": "gzip, deflate, br, zstd",
    "Accept-Language": "de,de-DE;q=0.9,en;q=0.8,en-GB;q=0.7
    "Content-Length": "0",
    "Host": "httpbin.org",
    "Origin": "https://httpbin.org",
    "Referer": "https://httpbin.org/",
    "Sec-Ch-Ua": "\"Microsoft Edge\";v=\"123\", \"Not:A-Bra
    "Sec-Ch-Ua-Mobile": "?0",
    "Sec-Ch-Ua-Platform": "\"Windows\"",
    "Sec-Fetch-Dest": "empty",
    "Sec-Fetch-Mode": "cors",
    "Sec-Fetch-Site": "same-origin",
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64
    "X-Amzn-Trace-Id": "Root=1-6629138a-30ccdbda63321bda3a8
  },
  "json": null,
  "origin": "46.127.119.188",
  "url": "https://httpbin.org/post"
}
```

Practical value

This kind of wrapper is especially helpful when you want WinInet's Windows integration without writing verbose API code. In practice, it gives Delphi developers a cleaner, component-based way to work with web services while staying close to native Windows networking.stackoverflow+1

```
const URL_APILAY_GEOCOUNTRY= 'https://api.api-ninjas.com/v1

function APIGetGeoCountry(AURL,url_name,aAPIKey: string): s
var encodedURL:String;
```

```
  begin
    encodedURL:= Format(AURL,[urlencode(url_name)]);
    with THttpRequestC.create(self) do
      try
        writ('reqsend: '+encodedurl)
        headers.add('X-Api-Key:'+aAPIkey);
        if Get(encodedURL) then
           result:= (Response.ContentAsString)
        else Writeln('APIError '+inttostr(Response.StatusCoc
      except
        writeln('HTTPS: '+ExceptiontoString(exceptiontype,ex
      finally
        free;
      end;
  end;
```

Here is the usual way to install HttpComponent in maXbox5: you don't have to install it cause its already in the core library in-built. The repository shows the component being created directly in code, for example `THttpRequestC.Create(self)`, then configured with headers before calling `Get`, `Post`, or `Post1Multipart`. That means you usually do not "install" it like a classic IDE package; instead, you add the source and use it as a script-ready Delphi component wrapper.

```
procedure SIRegister_THttpRequestC(CL: TPSPascalCompiler);
begin
  //with RegClassS(CL,'TComponent', 'THttpRequest') do
  with CL.AddClassN(CL.FindClass('TComponent'),'THttpReques
  begin
   Constructor Create( AOwner : TComponent)');
   Procedure Free');
    Function Delete( AUrl : String) : Boolean;');
    Function Delete1( AUrl : String; ABody : TBody) : Boole
    Function Delete2( AUrl : String; ABody : String) : Bool
    Function Get( AUrl : String) : Boolean');
    Function Head( AUrl : String) : Boolean');
    Function Options( AUrl : String) : Boolean');
    Function Patch( AUrl : String) : Boolean;');
    Function Patch1( AUrl : String; ABody : TBody) : Boolea
    Function Patch2( AUrl : String; ABody : String) : Boole
    Function Post( AUrl : String) : Boolean;');
    Function Post1( AUrl : String; ABody : TBody) : Boolean
    Function Post1Multipart( AUrl : String; ABody : TBody)
    Function Post2( AUrl : String; ABody : String) : Boolea
    Function Put( AUrl : String) : Boolean;');
    Function Put1( AUrl : String; ABody : TBody) : Boolean;
    Function Put2( AUrl : String; ABody : String) : Boolean
    Function Trace( AUrl : String) : Boolean');
    RegisterProperty('Cookies', 'TCookiesC', iptr);
   Cookies', 'TCookiesC', iptr);
   Response', 'THttpResponseC', iptr);
   Headers', 'THeaders', iptrw);
   UseCookies', 'Boolean', iptrw);
   AutoRedirect', 'Boolean', iptrw);
```

```
    SecurityOptions', 'TSecurityOptions', iptrw);
    UserAgent', 'String', iptrw);
    HttpVersion', 'THttpVersionComp', iptrw);
    Username', 'String', iptrw);
    Password', 'String', iptrw);
    OnProgress', 'THttpOnProgress', iptrw);
  end;
end;

THttpResponseC = class(TComponent)
  private
    FStatusCode: THttpStatus;
     FStatusCode2: cardinal;
    FHeaders: THeaders;
    FData: TBytes;
    function GetContentLength: Integer;
    function GetContentType: String;
    function GetContentAsString: String;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure SaveToFile(AFileName: String);
    property StatusCode: THttpStatus read FStatusCode;
     property StatusCode2: cardinal read FStatusCode2;
    property Content: TBytes read FData;
    property ContentAsString: String read GetContentAsStrir
    Property ContentAsAnsiString', 'String', iptr);
    Property ContentAsUTF8String', 'String', iptr);
    property ContentType: String read GetContentType;
    property ContentLength: Integer read GetContentLength;
    property Headers: THeaders read FHeaders;
  end;
```

Common Pitfalls

The main pitfalls with HttpComponent in maXbox5 come from WinInet's quirks (timeouts, HTTPS, encoding) plus scripting-environment issues like blocking calls and object lifetime. Below are the most common problems and practical ways to avoid them when you call the component from maXbox scripts.github+2

Initialization and lifetime

- Pitfall: Creating `THttpRequestC` as a local variable without proper `try..finally` can leak WinInet handles and streams if an exception is raised.theroadtodelphi+1
  Avoid it by using the standard Delphi pattern: create the request once, use `try..finally` to free it, even in scripts:text`var Req:`

```
THttpRequestC; begin Req := THttpRequestC.Create(nil);
try // use Req.Get/Post... finally Req.Free; end; end;
```

- Pitfall: Reusing the same request object and forgetting to clear headers or body between calls can send stale data (old cookies, custom headers, or form fields) to the next URL.github+1
  Avoid it by explicitly clearing or reinitializing state before each request (e.g. `Request.Headers.Clear;`, recreate the body object, or create a fresh request instance for logically separate calls).

Blocking calls in maXbox scripts

- Pitfall: `Get` and `Post` are synchronous; long requests block the maXbox interpreter, making the UI appear "frozen" during network delays.softwareschule+1
  Avoid it by:
  - Keeping calls short (local test URLs, small payloads) when running interactively.
  - Using timeouts on the WinInet side if exposed, or at least informing the user via a progress message before long operations.
- Pitfall: Doing multiple HTTP calls in a tight loop without `Sleep` or any UI feedback can make it hard to stop or debug the script.softwareschule
  Avoid it by adding small delays or logging (`writeln`, `ShowMessage`) in loops and testing with small batches first.

Encoding and response content

- Pitfall: Assuming all responses are ANSI and directly showing `Response.ContentAsString` in dialogs can corrupt UTF-8 or non-Latin text.theroadtodelphi+1, theres also a ContentAsUTF8String', 'String', iptr); and ContentAsAnsiString', 'String', iptr); available.
  Avoid it by:
  - Checking the `Content-Type` header (e.g. `charset=utf-8`) and converting bytes using the proper encoding if HttpComponent does not already do so for you.
  - Testing with known UTF-8 endpoints (like httpbin) and verifying characters display correctly.github
- Pitfall: Treating binary responses (images, PDFs, ZIPs) as strings can

lead to garbage display or lost data.softwareschule+1
Avoid it by:

- For binary data, saving `Response.ContentStream` (or equivalent) to a file or memory stream and processing it as binary rather than converting to text.github+1

HTTPS, certificates, and proxies

- Pitfall: HTTPS requests fail silently or return generic errors on older Windows systems if SSL/TLS support or flags are not set correctly in WinInet.theroadtodelphi+1
  Avoid it by:
  - Testing with simple `https://httpbin.org/get` and checking `Response.StatusCode` and error messages.github
  - Ensuring the underlying OS has up-to-date root certificates and TLS support; WinInet relies on the system's SSL stack.theroadtodelphi
- Pitfall: Requests fail in environments with HTTP proxies or strict firewalls because WinInet by default may use system proxy settings, which can be misconfigured.theroadtodelphi+1
  Avoid it by:
  - Testing first with a browser on the same machine and URL.
  - If your wrapper exposes proxy options, set them explicitly; otherwise verify the system proxy settings in Internet Options.theroadtodelphi

File upload and stream management

- Pitfall: When using multipart uploads (`AddFromFile` or a stream body), forgetting to set `ReleaseAfterSend` or to free the body object leads to memory and handle leaks in long-running scripts.theroadtodelphi+1
  Avoid it by:
  - Setting `Body.ReleaseAfterSend := True` when you do not need the body after `Post`, as in the official example.github
  - Otherwise, freeing any `TUrlEncodedFormBody` / multipart body in a `try..finally` block.
- Pitfall: Using incorrect or relative file paths when attaching files in maXbox scripts causes silent failures (e.g. "file not found" before the

HTTP request).softwareschule

Avoid it by:

- Using full paths (`ExpandFileName`) or ensuring the script's working directory is what you expect. Also set the right quotes to pass an URL, especially in Python4Delphi.
- Logging the final path you pass to `AddFromFile` to confirm it exists.

Error handling and diagnostics

- Pitfall: Only checking the boolean result of `Get`/`Post` and ignoring `Response.StatusCode` and message text makes debugging very hard.theroadtodelphi+1

  Avoid it by:

  - Always inspecting `Response.StatusCode` and showing or logging it:text`if Req.Get(URL) then writeln('OK: ', Req.Response.StatusCode) else writeln('Error: ', Req.Response.StatusCode, ' ', Req.Response.StatusText);`
  - For unexpected failures, print both the status code and any response body for server-side error details.github
- Pitfall: Treating network or protocol errors as generic exceptions without surfacing the underlying WinInet error text.theroadtodelphi+1

  Avoid it by:

  - Wrapping calls in `try..except` and logging `Exception.Message`.
  - If the wrapper exposes WinInet error strings, include them in your log output for easier diagnosis.theroadtodelphi

maXbox-specific integration issues

- Pitfall: Assuming you must "install" HttpComponent as a design-time package; in maXbox it is just source you use in scripts, so missing units cause "file not found" or "unit not found" compiler errors.github+1 dont bother you.

  Avoid it by:

  - Placing the HttpComponent units in a folder that maXbox5

knows (its script/source path) and checking the path settings in maXbox's configuration.softwareschule+1
    - Starting with a minimal example from the README (simple GET) before integrating into larger scripts.github
- Pitfall: Mixing maXbox's own WinInet helpers or Indy components with HttpComponent in the same script can make it unclear which one fails or how cookies/sessions are handled.torry+1
  Avoid it by:
    - Using one HTTP abstraction per script where possible (either maXbox's built-in helper or HttpComponent) until things work reliably.
    - If you must mix, keep them in clearly separated functions and log which one is used per call.

If you share a short maXbox5 script using HttpComponent by email, I can point out concrete pitfalls in that code and show "before/after" improvements tailored to your current usage.

Here are practical advanced usage patterns for `THttpRequestC` in maXbox5, based on how the wrapper is used for GET, POST, multipart upload, and response handling.github+1

Session patterns

A good advanced pattern is to create one request object per logical transaction, set headers explicitly, then dispose of it in `try..finally`. This keeps cookies, headers, and bodies from bleeding into later calls.github+1

Example structure:

```
textvar Req: THttpRequestC;
begin
  Req:= THttpRequestC.Create(nil);
  try
    Req.Headers.Add('Accept: application/json');
    if Req.Get('https://example.com/api') then
      writeln(Req.Response.ContentAsString);
  finally
```

```
    Req.Free;
  end;
end;
```


The wrapper's repository examples show this kind of direct object-based usage rather than design-time configuration.github

JSON POST

For API calls, send JSON by setting the `Content-Type` header and writing the JSON body as text. This is the cleanest pattern for REST endpoints that expect raw JSON rather than form fields.github

Example:

```
textReq.Headers.Add('Content-Type: application/json');
Req.Body:=
TStringStream.Create('{"name":"Max","mode":"test"}');
try
  Req.Post('https://api.example.com/items');
finally
  Req.Body.Free;
end;
```

The important part is matching the body format to the header, since servers often reject requests when those do not agree.qlik+1

Multipart upload

For file uploads, use the multipart body support and add files with `AddFromFile` as shown in the repository. This is the preferred route for forms that require both text fields and files.github+1

Practical tips:

- Use absolute paths for files.
- Confirm the file exists before calling `AddFromFile`.
- Set `ReleaseAfterSend := True` if the body should be freed

automatically after the request.github

Headers and auth

You can send custom headers such as `Authorization`, `Accept`, and `User-Agent` to adapt the request to different APIs. This is essential for services that require bearer tokens or strict content negotiation.qlik+1

A common pattern is:

```
textReq.Headers.Add('Authorization: Bearer ' + Token);
Req.Headers.Add('Accept: application/json');
```

If the server expects authentication, be careful to send the exact header format it requires.qlik

Response handling

Advanced use means checking more than just success or failure. Inspect `StatusCode`, `StatusText`, and the returned body so you can distinguish server errors from transport errors.qlik+1

A robust pattern is:

```
textif not Req.Get(URL) then
  writeln('Request failed: ', Req.Response.StatusCode)
else
  writeln('Body: ', Req.Response.ContentAsString);
```

If the response is binary, save it as a stream or file instead of converting it to

text.github

```
function TAddressGeoCodeOSM8(AURL, location, aApikey: strir
var Httpreq: THttpRequestC; httpres: string;
    Body: TMultipartFormBody;
    jsn: TMcJsonItem;
begin
  httpreq:= THttpRequestC.create(self);
  httpreq.headers.add('Accept: application/json; charset=ut
  httpreq.useragent:= USERAGENT5;
  httpreq.SecurityOptions:= [soSsl3, soPct, soIgnoreCertCNI
  try
    if httpreq.get(Format(AURL,[location])) then begin
       httpres:= (httpreq.Response.ContentAsString)
       //writeln('conttype '+httpreq.Response.ContentType);
       writ('debug back '+formatJson(httpres));
       jsn:= TMcJsonItem.Create;
       jsn.AsJSON:= httpres;
       result.lat:= jsn.at(0,'lat').asnumber;
       result.long:= jsn.at(0,'lon').asnumber;
       result.descript:= Format('Coords: lat %2.5f  lng %2.
                                [result.lat,result.long,jsr
                                                      jsr
    end else Writeln('APIError '+inttostr(Httpreq.Response.
    //StrReplace(httpres, '[{', '{');
  finally
    writeln('Status3: '+gethttpcod(httpreq.Response.statusc
    httpreq.Free;
    sleep(200)
    jsn.Free;
  end;
end;
```

maXbox scripting style

In maXbox5, keep scripts short, linear, and debuggable because synchronous HTTP calls can block the interpreter. For more complex flows, split logic into small functions: one for setup, one for sending, and one for checking the response.softwareschule

A useful debugging trick is to log the URL, headers, and payload before each call so you can compare the script's behavior with the server's expectations. That usually makes failures much easier to isolate than stepping through a long script.softwareschule

Would you like a **full maXbox5 example script** for GET, JSON POST, and multipart upload using `THttpRequestC`?

maXbox5 – Manage /examples at SourceForge.net

Batch Requests

`THttpRequestC` does not appear to have built-in “batch” support; the practical pattern is to loop over a list of URLs or payloads and create one request per item, reusing the same component only if you carefully reset its state between calls. In maXbox, the safest approach is usually a simple sequential loop with `try..finally` around the request object.github+2

Recommended pattern

Use one request object, send one request, read the response, then move to the next item. That mirrors how HTTP is normally used and avoids stale headers, bodies, or cookies leaking from one call to the next.stackoverflow+1

```
textvar
  Req: THttpRequestC;
  i: Integer;
  Urls: array[0..2] of string;
begin
  Urls[0] := 'https://httpbin.org/get';
  Urls[1] := 'https://httpbin.org/headers';
  Urls[2] := 'https://httpbin.org/ip';

  Req := THttpRequestC.Create(nil);
  try
    Req.Headers.Add('Accept: application/json');

    for i := Low(Urls) to High(Urls) do
    begin
      if Req.Get(Urls[i]) then
        Writeln(Urls[i] + ' -> ' +
IntToStr(Req.Response.StatusCode))
      else
        Writeln(Urls[i] + ' -> ERROR ' +
IntToStr(Req.Response.StatusCode));
    end;
  finally
    Req.Free;
  end;
end;
```

Things to watch

Keep requests sequential unless you build your own threading, because maXbox scripts are synchronous and a long network call can block the interpreter. Also, if you send different request types in the same loop, reset or rebuild the body object each time so previous POST data does not get reused accidentally.softwareschule+1

If you need many calls for a larger batch, organize the inputs in an array, file, or string list, then iterate and log each result separately. HTTP itself is stateless, so each request should be treated as independent unless you intentionally depend on cookies or server session state.stackoverflow

Posted in Code, Models, TEE, Tools

One response to “WorldNewsAPI”

Max Kleiner
April 16, 2026 Edit

Direct code:

softwareschule.ch/examples/worldnewsapi.txt

softwareschule.ch/examples/worldnewsapi.htm

★ Like

Reply