Overview

TMS Sparkle is a Delphi framework for network, Internet programming. It provides classes for both client and server applications/services, allowing performing HTTP requests to servers, or building HTTP servers that receive and process client requests. It supports several platforms, including Microsoft Windows, Linux, Mac OS X, iOS and Android. TMS Sparkle is also the core framework used by several other TMS products such as TMS XData and TMS RemoteDB.

TMS Sparkle product page: https://www.tmssoftware.com/site/sparkle.asp

TMS Software site: https://www.tmssoftware.com

Why TMS Sparkle?

- **Trustworthy**: It is the core building block for several other TMS products and technologies, such as TMS RemoteDB and TMS XData. Such products needed to be built from scratch, and rely on a robust framework for which TMS could have 100% control and also responsibility. For such products to work flawlessly, we needed to be sure to build such products in a framework that must be properly tested, and have fast response in performance improvement and bug fixing.
- **Fresh**: It's a new product that doesn't carry any legacy applications behind it, thus classes and interfaces provide a simple, clean method of use.
- **Modern**: It's target to new Delphi versions, and benefits from modern language features such as generics and anonymous methods.
- **Cross-platform**: Supports multiple platforms such as Microsoft Windows, Linux, Mac OS X, iOS and Android.
- **Platform Native**: In most of it, Sparkle is a thin, abstract layer over native API's from the underlying platform. Existing platforms already provide a lot in terms of native networking and internet programming. TMS Sparkle tries not to reinvent the wheel and use such technologies. This makes it easy for your applications to benefit from new platform versions and upgrades. Any bug fixes and improvements in the platform frameworks will be usually available in Sparkle. It also provides smoother integration with the platform, such as system-wide settings.

Client features

- HTTP client available in Windows, Linux, Mac OS X, iOS and Android;
- Support for HTTP Secure (HTTPS);
- Transparent handling of chunked and gzip-encoded responses.

Server features

- Windows HTTP server based on http.sys stack (minimum Vista and Server 2008);

- Linux HTTP server based on WebBroker Apache module (requires Delphi 10.2 Tokyo or later);
- Support for HTTP Secure (HTTPS);
- Kernel-mode caching and kernel-mode request queuing on Windows (less overhead in context switching);
- Multiple applications/process can share (respond) the same port (at different addresses);
- Secure Sockets Layer (SSL) support in kernel-mode.

In this section:

Http Client
Performing HTTP client requests from your application.

Http Server
Building a server to process and respond to HTTP requests.

Design-Time Components
TMS Sparkle components for design-time usage.

Middleware System
Creating middleware interfaces in order to pre/post-process requests.

Authentication and Authorization
Built-in mechanisms for authentication and authorization.

Built-in Modules
Available and ready-to-use TMS Sparkle modules.

JSON Classes
Classes for manipulation JSON representation of data.

Logging
The logging system from TMS Business.

Http Client

Sparkle provides classes to perform HTTP client requests from your application. Basic usage is simple:

1. Create an THttpClient class (declared in `Sparkle.Http.Client` unit);
2. Call CreateRequest to create a THttpRequest object;
3. Fill the request object properties;
4. Call Send method passing the request to receive a THttpResponse object;
5. Use the response object properties to inspect the response.

The THttpClient class is available in several platforms: Windows, Mac OS X, Android and iOS (iPad/iPhone). You can use either http or https addresses.

The following example sends a post request to address *http://myserver/customers*, passing a string as the request body, check if the response status code is 200, and if it does, get the response content body as string.

```
uses {...}, Sparkle.Http.Client;

var
  Client: THttpClient;
  Request: THttpRequest;
  Response: THttpResponse;
  ResponseBody: string;
begin
  Request := nil;
  Response := nil;
  Client := THttpClient.Create;
  try
    Request := Client.CreateRequest;
    Request.Uri := 'http://myserver/customers';
    Request.Method := 'POST';
    Request.SetContent(TEncoding.UTF8.GetBytes('Request content'));
    Response := Client.Send(Request);
    if Response.StatusCode = 200 then
      ResponseBody := TEncoding.UTF8.GetString(Response.ContentAsBytes);
  finally
    Request.Free;
    Response.Free;
    Client.Free;
  end;
```

The following topics describe more details about using the HTTP client class.

Configuring a Request

After you use the THttpClient to create a THttpRequest, there are several properties you can use to properly configure your request before sending it to the server.

Defining the URI

Use Uri property to specify the server URI to send the request:

```
Request.Uri := 'http://myaddress.com';
```

Defining the request method

Use the Method property to specify the request method:

```
Request.Method := 'DELETE';
```

Specifying request headers

You can use Headers property to define custom headers for the HTTP request. The Headers property provides you with a THttpHeaders object. There are several methods you can use in this headers object to manipulate headers. The following example clears headers and set the value of ETag header.

```
Request.Headers.Clear;
Request.Headers.SetValue('ETag', '"737060cd8c284d8af7ad3082f209582d"');
```

Defining request content body

Use SetContent method to set the content of the request to be sent to server. The SetContent method receives a byte array as parameter.

```
Request.SetContent(TEncoding.UTF8.GetBytes('content'));
```

Setting request timeout

Use Timeout property to specify the maximum time length (in milliseconds) the client will wait for a response from the server until an error is raised. Default value is 60000 (60 seconds).

```
Request.Timeout := 30000; // set timeout to 30 seconds
```

Examining the Response

After sending a request to the server using the THttpClient, you will receive a THttpResponse object. There are several properties you can use to examine the response returned by the server. You are responsible for destroying the THttpResponse after you use it. You can also put the instance in a variable decalred as IHttpResponse (which has the same properties as THttpResponse) to benefit from interface automatic reference counting.

Status code

Use StatusCode property to examine the code returned by the server.

```
if Response.StatusCode = 200 then // Ok!
```

Response Headers

Use Headers property to examine the headers returned in the HTTP response. The Headers property provides you with a THttpHeaders object. There are several methods you can use in this headers object to manipulate the headers.

Content Length

Property ContentLength gives you the length of the content returned by the server. If the content was gzip encoded, this property will give you the value for the size of content already decompressed.

Content Type

Use ContentType property to examine the value of content-type returned by the server.

ContentAsBytes

To read all the response body at once, you can use ContentAsBytes property. This will return the whole body in a byte array.

ContentAsStream

ContentAsStream property will return a TStream object representing the content of response body. You can read from stream to receive content from server. If the response is chunked, the client might still be receiving data from the server while you read from the stream. If you reach the end of stream and data transfer is not completed yet, a read operation on the stream will wait until more data is available in the stream. Never use the Size property of the stream since the stream grows dynamically as you keep reading from it. Use ContentLength to know the exact size of the stream. If the response is chunked, ContentLength will be 0 and you must keep reading from the stream until the returned number of bytes in a Read operation is zero.

Chunked property

Indicates if the response content is sent in chunked encoding (true).

THttpHeaders object

The THttpHeaders object provides several methods for you to read and set headers of either a response or request object. It's used for both client and server classes in Sparkle. The following are a list of main properties and methods available. The THttpHeaders object is declared in unit `Sparkle.Http.Headers`.

For all methods, the header names are **case-insensitive**, so a call to *Get('Accept')* is equivalent to *Get('accept')* or *Get('ACCEPT')*.

Setting a header value

Call SetValue method to set the value of a header. You must pass the header name, and header value. If the header name already exists, the current value will be replaced by the new one.

```
Headers.SetValue('ETag', '"737060cd8c284d8af7ad3082f209582d"');
```

Retrieving a header value

Use Get method to retrieve the value of a header. You **don't** need to check if a header exists before trying read its value. If you try to execute the Get method passing a header that doesn't exist in the request, an empty string will be returned. For a more precise way of checking if the header exists in a request or, you can use Exists method.

```
ContentType := Headers.Get('Content-Type');
```

Checking if a header exists

Use Exists method to verify if a header exists. This is useful to know if a header is present in a request or response message.

```
HasAcceptHeader := Headers.Exists('Accept');
```

Iterating through all headers

You can also iterate through all headers in the request/response by using AllHeaders property. The result type of this property is TEnumerable<THttpHeaderInfo>. The THttpHeaderInfo is just a record with both header name and value:

```
uses {...}, Sparkle.Http.Headers;

var
  Info: THttpHeaderInfo;
begin
  for Info in Headers.AllHeaders do
    // Use Info.HeaderName and Info.HeaderValue to retrieve header name and
value, respectively
```

Removing a header

Call Remove method to remove a header from message:

```
Headers.Remove('Accept');
```

Clearing headers

You can use Clear method to clear all defined headers.

```
Headers.Clear;
```

THttpClient Events

THttpClient class has events to help you control the client/server communication.

OnSendingRequest event

```
THttpRequestProc = reference to procedure(ARequest: THttpRequest);

property OnSendingRequest: THttpRequestProc;
```

This event is called right before a request is sent to the server. It's an opportunity to inspect the THttpRequest object and do some last-minute modifications, like for example adding a header common to all requests, or doing some logging of requests being sent.

```
MyHttpClient.OnSendingRequest :=
  procedure(Req: THttpRequest)
  begin
    Req.Headers.SetValue('custom-header', 'customvalue');
  end;
```

OnResponseReceived event

```
THttpResponseProc = reference to procedure(ARequest: THttpRequest; var AResponse:
THttpResponse);

property OnResponseReceived: THttpResponseProc;
```

This event is called right after a response is received from the server. It's an opportunity to inspect the THttpResponse object and do some generic processing. The AResponse object is passed by reference meaning you can replace it by another one in case you want to alter the response for further processing of the framework. If you do this, you must destroy the previous AResponse object.

Proxy configuration on Windows

When on Windows, you can configure the proxy used for connections. There are three modes for using proxies:

Default

This is the default mode. Sparkle http client on Windows is based on WinHttp library. When proxy is set to this mode, Sparkle will use the default proxy settings for WinHttp library. Note that this is not the default proxy used by Internet Explorer. The proxy for WinHttp is set using specific code, using netsh command-line (you can find an example here: Netsh Commands for WINHTTP).

Custom

In this mode, it's you that manually define the proxy address.

Auto

Proxy settings will be detected automatically based on current Windows settings (Internet Explorer and other global settings). Supported on Windows 8.1 and later only. If your application is running on a Windows version below 8.1, the mode will automatically switch to Default mode.

The following code illustrates how to use each mode, from an existing THttpClient instance (represented here by FClient variable):

```
uses {...}, Sparkle.WinHttp.Engine;

var
  Engine: TWinHttpEngine;
begin
  Engine := TWinHttpEngine(FClient.Engine);

  // Option 1: Current behavior
  Engine.ProxyMode = THttpProxyMode.Default;

  // Option 2: Get proxy settings automatically (windows 8.1 and later only)
  Engine.ProxyMode := THttpProxyMode.Auto;

  // Option 3: Custom proxy settings
  Engine.ProxyMode := THttpProxyMode.Custom;
  Engine.ProxyName := 'localhost:8888';

  // Force a new session to use new proxy settings
  Engine.ResetSession;
end;
```

Bypassing Self-Signed Certificates on Windows

By default Windows HTTP client raises an error if you try to connect to a server that has a wrong certificate, like wrong date, domain name, etc. You can bypass this protection and allow the client to connect. This is usually useful when you want to test your client against a server with a self-signed certificate. But beware that this could create a security issue by allowing that!

For that, you need to use units `Sparkle.WinHttp.Engine` and `Sparkle.WinHttp.Api` and then add an event handler to the WinHttp engine. In the example below, FClient is of type THttpClient.

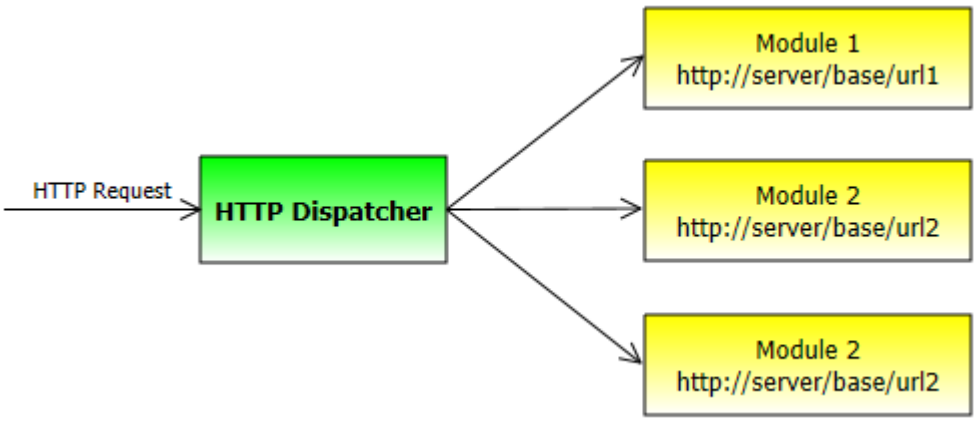
```
uses
  {…}, Sparkle.WinHttp.Engine, Sparkle.WinHttp.Api;


  // FClient is of type THttpClient
  TWinHttpEngine(FClient.Engine).BeforeWinHttpSendRequest :=
    procedure(Handle: HINTERNET)
    var
      dwFlags: DWORD;
    begin
      dwFlags := SECURITY_FLAG_IGNORE_UNKNOWN_CA or
        SECURITY_FLAG_IGNORE_CERT_WRONG_USAGE or
        SECURITY_FLAG_IGNORE_CERT_CN_INVALID or
        SECURITY_FLAG_IGNORE_CERT_DATE_INVALID;
      WinHttpCheck(WinHttpSetOption(Handle, WINHTTP_OPTION_SECURITY_FLAGS, @dwFla
gs, SizeOf(dwFlags)));
    end;
```

Http Server

Sparkle provides classes for you to build an Http Server to process and respond to http requests. In this chapter we will explain how these classes work and how you can build your Http Server using Sparkle.

Overview

Sparkle Http Server provides three concepts, a server, a dispatcher and a server module (or simply "module").



The server is higher-level object that handles the communication process to you. Currently you can have three different server types: Http.Sys-based server, Apache-based server, Indy-based server and in-process server.

Each server is different from each other, but all them have a common object: the HTTP Dispatcher. The dispatcher is the object that holds information about all server modules and handles the requests.

Finally a module is a logical part of the whole server architecture that you register into the dispatcher to receive requests for a particular URI address. Each module has an URI associated with it, and the dispatcher is responsible to redirect the request to the proper module, based on the requested URI. A module receives requests for the URI he's associated with, and also for all other URI which parent is the associated URI.

For example, suppose you have registered two modules 1 and 2, and their associated URIs are "http://server/base/url1" and "http://server/base/url2", respectively.

Module 1 will receive requests and provide responses for URIs like

```
http://server/base/url1/
http://server/base/url1/test/
http://server/base/url1?q=1
```

While Module 2 will receive requests and provide responses for URIs like

```
http://server/base/url2/
http://server/base/url2/page.htm
http://server/base/url2/page2.html\#43
```

Note that the dispatcher will forward the request to the proper module based on the base URI, but then the module must be responsible to parse the URI to provide the correct response.

Wizard For New Sparkle Server

You can use the "New TMS Sparkle Server" wizard to create a new server with a few clicks. The wizard will only create Http.sys-based server. You can also freely create the server manually, the wizard is not mandatory and it's just a way to get started quickly. For Apache-based and in-process server, no wizard is available and you have to create it manually.

To create a new Sparkle Server:

1. chose **File > New > Other** and then look for "**TMS Business**" category under "Delphi Projects". Then double click "**TMS Sparkle Server**".
2. Chose the kind of applications you want to server to run on, then click *Next*.
   Available options are **VCL**, **FMX**, **Service** and **Console**. You can choose multiple ones (for example, you can have a VCL project for quick test the server and a Service one to later install it in production, both sharing common code.
3. Chose the **Host Name**, **Port** and **Path** for the server, then click *Create*.
   Usually you don't need to change the host name, for more info check URL namespace and reservation. Port and Path form the rest of base URL where your server will be reached.

The new server will be create and ready to run. The Sparkle Server template just responds any request with a "Hello, World" message.

Using Design-Time Components

Another way to use TMS Sparkle is by using the design-time components. If you want the RAD, component dropping approach, this is the way.

There is a full chapter about the design-time components, but here is a short summary about how to use it:

1. Drop a dispatcher component in the form (for example, TSparkeHttpSysDispatcher);
2. Drop a server component in the form (TSparkleStaticServer, TXDataServer, etc.);
3. Associated the server to the dispatcher through the Dispatcher property of server component;
4. Specify the BaseUrl property of the server (for example, *http://+:2001/tms/myserver*);
5. Configure any specific property of the server;
6. Set Active property of dispatcher component to true.

Http.Sys-based Server

For servers running on Windows platform, Sparkle provides an HTTP server based on Microsoft HTTP Server API, which is an interface for the HTTP protocol stack (HTTP.sys). This makes Sparkle HTTP Server a full-featured, robust and fast server on Microsoft Windows. Many operations like routing HTTP requests, caching of responses, encryption and authentication runs in kernel-mode. It's the same stack used by Microsoft IIS.

It plays the role of the "HTTP Server" as described in the overview of Sparkle server architecture. It's a Delphi class that is a layer over the Microsoft API, receive the requests from operational system, and dispatch the requests to the proper server modules.

The following code illustrates how to create the server, add modules and start it.

```
uses {...}, Sparkle.HttpSys.Server;

var
  Server: THttpSysServer;
begin
  Server := THttpSysServer.Create;
  Server.Dispatcher.AddModule(TMyServerModule.Create('http://host:2001/
myapplication');
  Server.Dispatcher.AddModule(TAnotherModule.Create('http://+:2001/anotherapp');
  Server.Start;

  // server runs in console mode until user hits Return
  ReadLn;

  Server.Stop;
end;
```

In the example above, any request to URIs beginning with "http://server.com/myapplication" will be dispatched to TMyServerModule. And any requests to address "http://host:2001/" will be dispatched to TAnotherModule.

By default, Sparkle will strip the host part of the url and replace with a + symbol. So even in the first example with "host" as the server, internally it will change to *+:2001*. This is desired for most cases. If you want Sparkle to keep the host name (for example, if you are listening to different IP address/host names in the same server, then you can set KeepHostInUrlPrefixes property to true:

```
Server.KeepHostInUrlPrefixes := True;
```

Since THttpSysServer is based on the http.sys stack, the URI you want to listen (like the "http://+:2001/" in example) must be reserved in the Windows. Otherwise your application won't be able to respond to such requests. This is not Sparkle-related, but all requirements on the http.sys itself. There is plenty documentation over the internet about how to reserve and configure many details of the server, like configuring server-side certificates, for example. However, Sparkle also provides helpful tools for you to accomplish these tasks. The following topics will try to summarize the basic steps you need to perform such operations and how Sparkle can help you out with it.

URL namespace and reservation

Sparkle THttySysServer is based on kernel-mode http.sys which means all http requests are processed by the operational system. In this architecture, the kernel forwards the http requests based on the requested url. For this to work you need to first reserve an Url namespace. Namespace reservation assigns the rights for a portion of the HTTP URL namespace to a particular group of users. A reservation gives those users the right to create services that listen on that portion of the namespace.

So, if all your servers will run under the address *http://server:2001/tms/*, you can reserve that namespace to make sure Windows will listen HTTP requests to those addresses instead of refusing them.

To reserve an Url namespace, you can either:

- TMSHttpConfig tool (to easily configure using GUI);
- THttpSysServerConfig class (to configure from Delphi code);
- Windows netsh command-line tool (to learn how to configure with Windows itself without using Sparkle).

The Url reservation just need to be done one time for the machine you are going to run the server. If you try a second time, an error will occur.

Using HTTP secure (HTTPS)

You can configure your server to work with HTTP secure. To do this, you need to previously bind an existing certificate to the port you are going to use for the HTTPS connection. This way, when the http.sys server receives an HTTPS request to a specified port, it will know which server certificate to send to the client.

Just as with URL reservation, there are several ways you can bind a certificate to a port:

- TMSHttpConfig tool (to easily configure using GUI);
- THttpSysServerConfig class (to configure from Delphi code);
- Windows netsh command-line tool (to learn how to configure with Windows itself without using Sparkle).

One you have bound the certificate to the port in server, using HTTPS is pretty straightforward with Sparkle. You don't need any extra SSL libraries to be installed/deployed either in client or server. All SSL communication is done native by the underlying operational system.

Use "https" prefix in Delphi code

Once you have registered your certificate with the command above, your server is configured to use secure connections. Please note that you still need to reserve the url for the connection, and the url must begin with "https" (for example, you might want to reserve the url namespace "https://+:2002/tms/business".

When registering modules in the HTTP server, all you need to do is provide the correct base URI that matches the reserved namespace. Don't forget that you must prefix the URI with "https".

Using a certificate for testing

If you don't have a certificate, you can still generate a self-signed certificate for testing purposes. For that you will need makecert.exe tool, which is available when you install either Microsoft Visual Studio or Windows SDK. Generating a self-signed certificate is out of scope of this documentation, but the following links might help in doing such task. Once you have generated and installed a self-signed certificate, the process for using it is the same as described previously, all you need is bind the certificate to the HTTPS port using the certificate thumbprint.

How to: Create Temporary Certificates for Use During Development

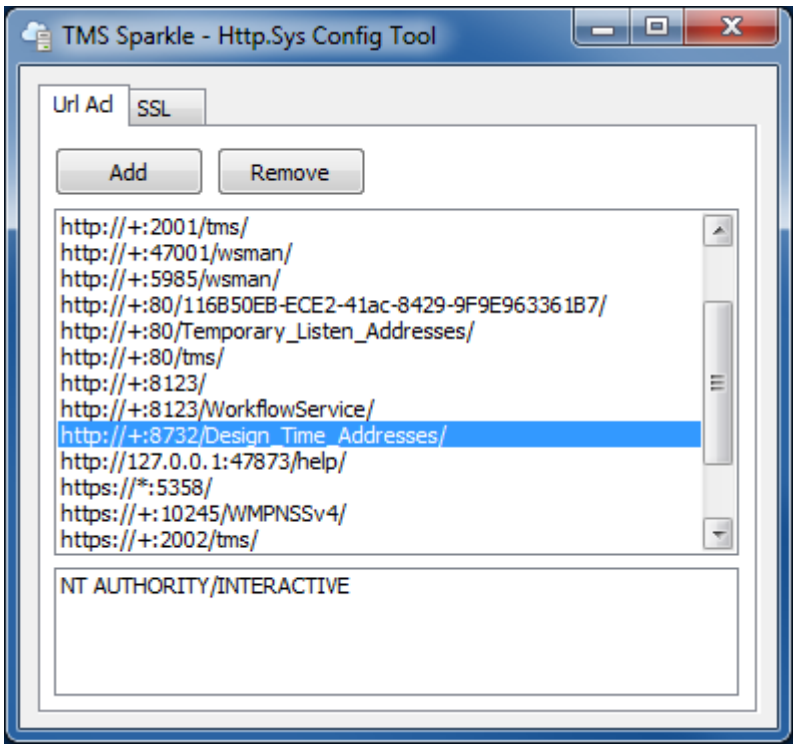
> **NOTE**
> Self-signed certificates are only for testing/development purposes. For production environment, use a certificate signed by a certificate authority.

TMSHttpConfig Tool

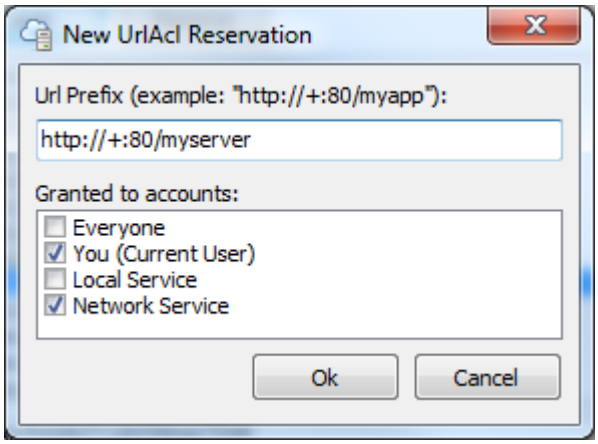
Using the TMSHttpConfig tool is the easiest way to configure *http.sys* on your server. TMS Sparkle distribution includes a binary executable of TMSHttpConfig which runs stand-alone and don't need to be installed nor needs any extra files to be executed. Just run TMSHttpConfig.exe in the computer where your server will run and use it. Also full source code of this tool is avaliable in the demos distribution of TMS Sparkle.

URL Reservation

TMSHttpConfig shows you all the existing *http.sys* URL reservations in the "Url Acl" tab. You can browse them and if you select a row in list, it will show you to which accounts the URL is reserved to (use is permitted).



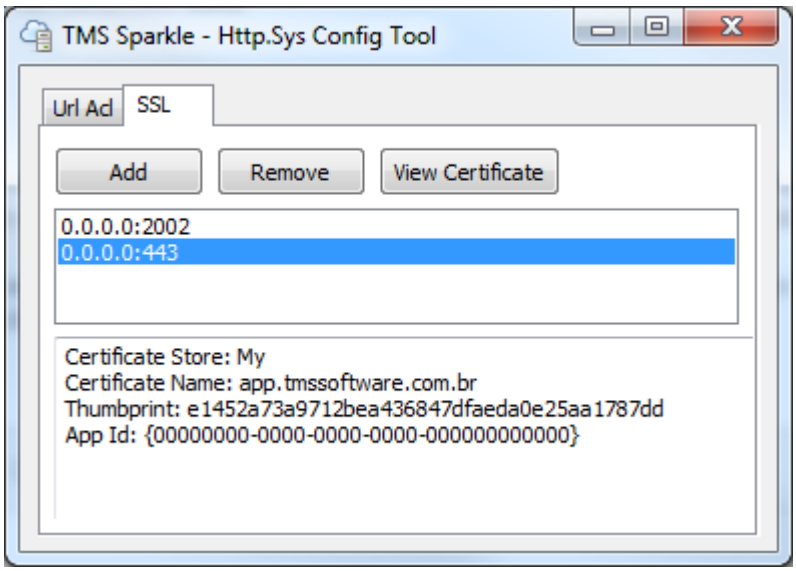
You can add a new URL reservation, or remove an existing one by using the Add and Remove buttons. To add a new reservation, you just need to type the URL prefix according to Microsoft rules. Usually just use the format "http://+:<port>/<basepath>". You must also choose to which Windows accounts the reservation will be added to. TMSHttpConfig predefines your account (for testing purposes and running stand-alone servers) and Network Service (to run from Windows services).



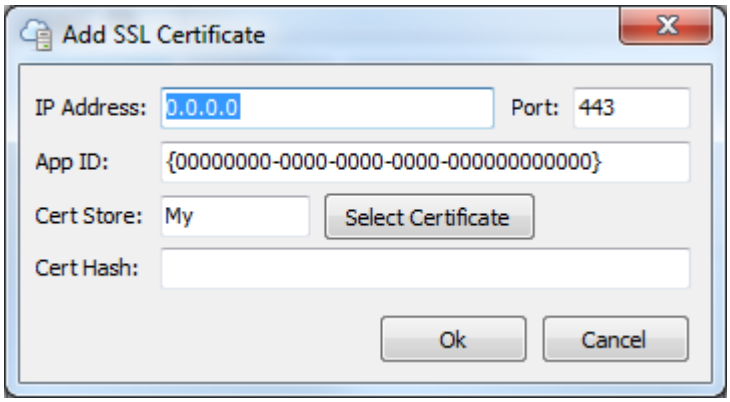
Server Certificate Configuration

It's also very easy to bind a server certificate to a port to use HTTPS with Sparkle servers. First, you must be sure your certificate is already installed/imported in the Windows Certificate Store. Also note that you must install it to Local Machine store, not the Current User. For more information about how to to this, follow this link. Usually your certificate provider will give you detailed instructions about how to do this.

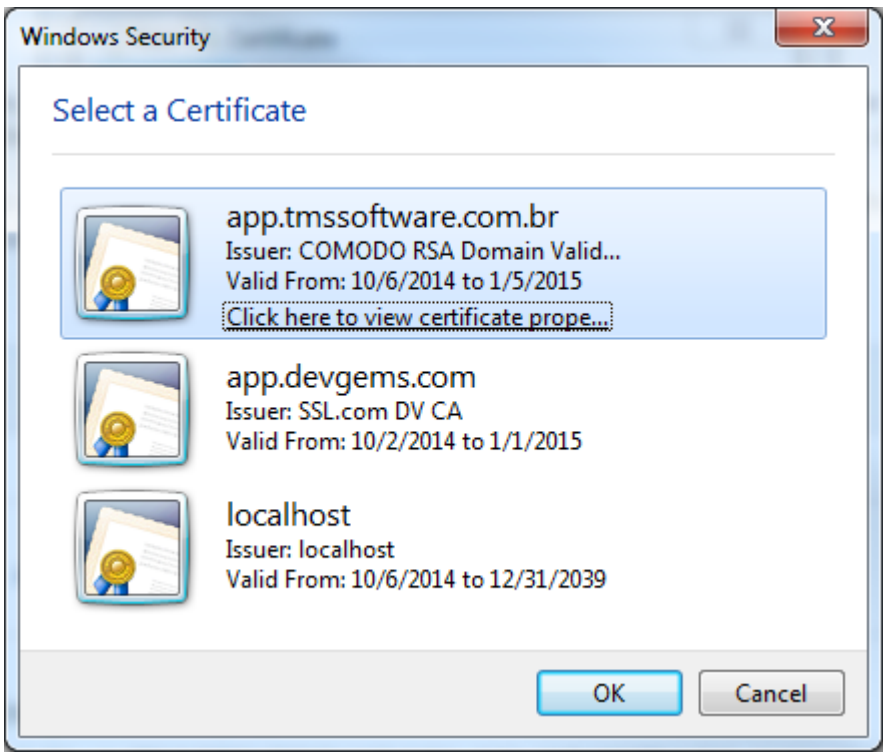
For a review, TMSHttpConfig shows you all existing port-certificate binds in the "SSL" tab. If you select a row, it will show you a summary about the binding below the list. You can see information about the certificate bound to the port, and the app id used for the binding.



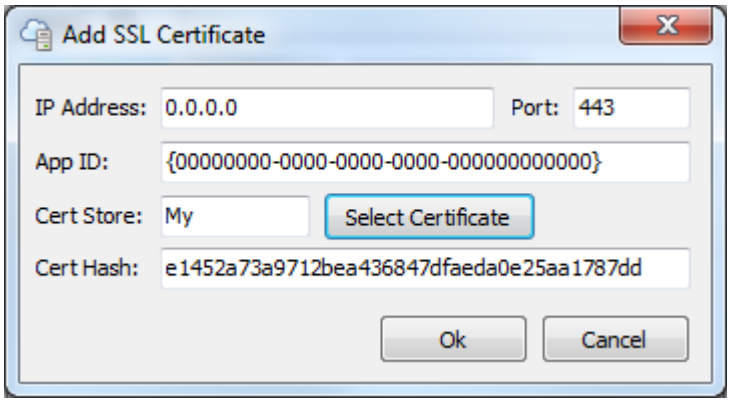
To add or remove a binding use "Add" and "Remove" buttons. When you click "Add", the following screen appears.



- *IP Address*: You should just leave it with default "0.0.0.0" value. In the case you rarely need to bind the certificate to a specific IP only, just type the IP. The default value will work for all IP addresses.
- *Port*: Type the port where the certificate will be bound to.
- *App ID*: This field is just for information and is not needed for the server to function. It must be a GUID (enclosed by brackets) and you can just leave the default empty GUID provided by TMSHttpConfig.
- *Cert Store*: Indicates which certificate store you will use to retrieve the certificate. Also, the default is "My" (Personal store) and that's where your certificate probably is so it's also unlikely you will need to change this field.
- *Cert Hash*: This field should contain the thumbprint (hash) of the certificate to be bound to the part. You could just type it here, but it's way easier to click "Select Certificate" button to do that. When you click that button, TMSHttpConfig will show you a list of all available certificates in the chosen certificate store:

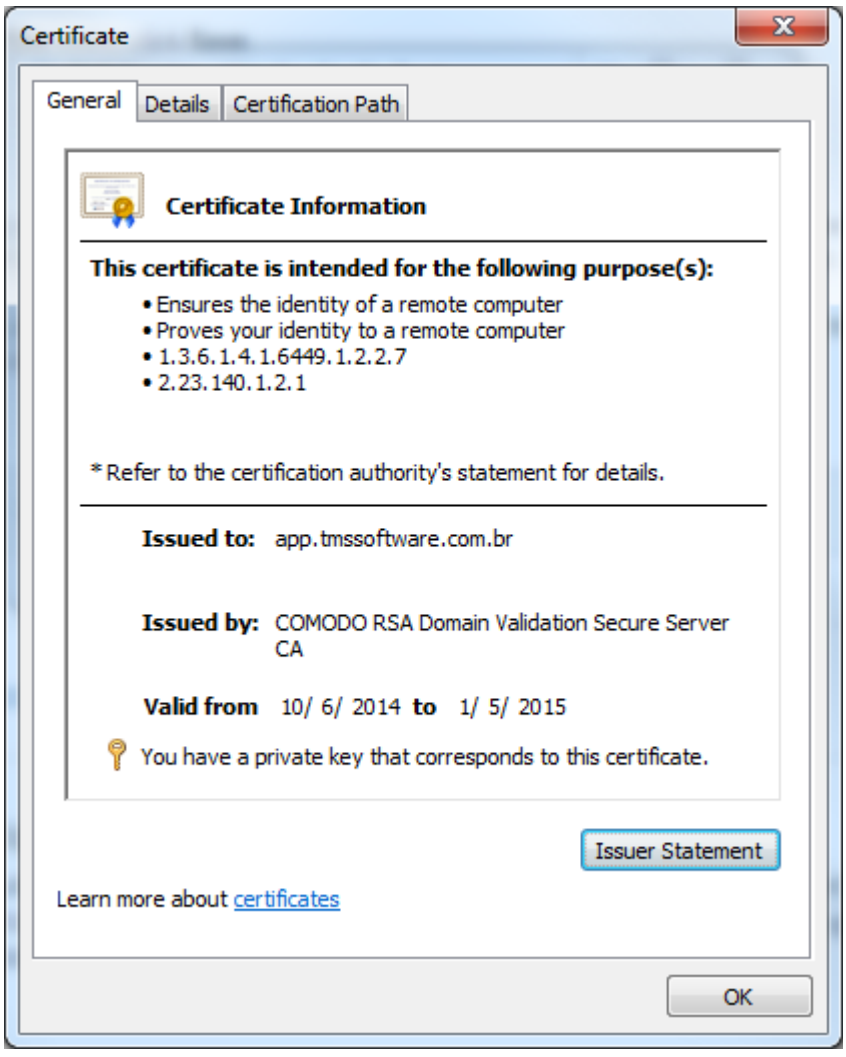


Just select your certificate, click "Ok", and the thumbprint will be filled automatically for you.



Once all fields are correct, just click "Ok" and the binding is done. Your server is now responding to HTTPS requests in the specified port using the specified certificate.

For later review, if you can select a binding in the list and click "View Certificate" to see more detailed information about the certificate bound to the port.



THttpSysServerConfig Class

To perform URL reservation and configure server certificate to use HTTPS, easier way is use the TMSHttpConfig tool. But it might be possible that you want to automate this registration procedures, or build your own tool to do that using Delphi. In this case, Sparkle provides the

THttpSysServerConfig class (declared in unit Sparkle.HttpSys.Config) that allows you to do all those tasks from Delphi code. Just remember that your application must be running with administrative rights.

All the following examples assume you have a Config variable with an existing object instance of a THttpSysServerConfig class:

```
uses {...}, Sparkle.HttpSys.Config;

var
  Config: THttpSysServerConfig;
begin
  Config := THttpSysServerConfig.Create;
  try
    // Do your config code here using Config instance
  finally
    Config.Free;
  end;
end;
```

URL Reservation

To reserve an URL from Delphi code using TMS Sparkle classes, use the following code.

```
if not Config.IsUrlReserved('http://+:2001/tms/business/') then
  Config.ReserveUrl('http://+:2001/tms/business/', [TSidType.CurrentUser, TSidTyp
e.NetworkService]);
```

You can remove a reservation using RemoveURL:

```
Config.RemoveUrl('http://+:2001/tms/business/');
```

To get a list of all existing URL reservations, use the Reservations enumeration property, which will retrieve you object instances of TUrlAclInfo class.

```
var
  Info: TUrlAclInfo;
begin
  Config.RefreshReservations;
  for Info in Config.Reservations do
    ReservedPrefix := Info.UrlPrefix;
end;
```

Server Certificate Configuration

To add a bind a certificate to a port, use AddSslCert method:

```
Config.AddSslCert('0.0.0.0', 443, 'My', Thumbprint, MyAppGuid);
```

First three parameter are IP, Port and Certificate Store. Fourth parameter is the certificate thumbprint in binary format (TBytes). Finally, last parameter is a TGUID for the App Id (you can use an empty GUID here). For more information about those parameters, see the description in TMSHttpConfig tool topic.

To remove a binding, using RemoveSsl method just passing IP and port:

```
Config.RemoveSsl('0.0.0.0', 443);
```

To get a list of all existing certificate bindings, use SslCerts enumeration property, which will give you object instances of TSslCertInfo class.

```
var
  Info: TSslCertInfo;
begin
  Config.RefreshSslCerts;
  for Info in FConfig.SslCerts do
    ListBox1.AddItem(Format('%s:%d', [Info.Ip, Info.Port]), Info);
  ListBox1.Sorted := true;
end;
```

The TMSHttpConfig tool uses THttpSysServerConfig under the hood to perform its actions. So for more detailed info about how to use these classes you can refer to its source code, available in Sparkle demos.

Windows netsh Command-Line

If you don't want to use neither TMSHttpConfig tool nor THttpSysServerConfig class, you can just use Windows command line tool *netsh* to do URL reservation and HTTPS config in your server. The following examples reserve the Url *http://+:2001/tms/business* (meaning we will be able to process requests to the port 2001, if the requested url starts with tms/business). Detailed info can be found in this Microsoft article.

URL Reservation

For this, just run windows command line tool (cmd.exe) *under administrative rights* and execute the following command:

```
netsh http add urlacl url=http://+:2001/tms/business/ user=%USERDOMAIN%\
%USERNAME%
```

where %USERDOMAIN% and %USERNAME% are the domain and name of the user under which your server will run. For testing purposes, you can just give access to any user:

```
netsh http add urlacl url=http://+:2001/tms/business/ user=Everyone
```

Note that if the language of your Windows is not English, you must need to change "Everyone" by the name of the group that represents all users in Windows. Or, alternatively, provide the ssdl of the "everyone" group (or any other group you want to give permission to, for example replace "WD" by "NS" to provide access to network service.

```
netsh http add urlacl url=http://*:2001/tms/business/ sddl=D:(A;;GA;;;WD)
```

Server Certificate Configuration (binding to a port)

Run windows command line tool (cmd.exe) *under administrative rights* and use a command like this:

```
netsh http add sslcert ipport=0.0.0.0:2002
certhash=0000000000003ed9cd0c315bbb6dc1c08da5e6 appid={00112233-4455-6677-8899-
AABBCCDDEEFF}
```

The above command will bind the proper certificate to port 2002. There are three parameters in the command above that you need to change for your own usage:

- *ipport*: You must use the port number you use for HTTPS connections. In the example, it was 2002. The IP can still be 0.0.0.0 which means any IP.
- *certhash*: You must provide the thumbprint of the certificate you want to use for your server. You can check the thumbprint by using Microsoft Management Console. Please refer to Microsoft article "How to: Retrieve the Thumbprint of a Certificate" for detailed information.
- *appid*: This can be any arbitrary GUID. You just need to generate one and input it here. You can even use the GUID generator in Delphi code editor (while editing code, just press Shift+Ctrl+G, Delphi will generate a GUID for you). The GUID must be enclosed by brackets. For more information, please refer to "How to: Configure a Port with an SSL Certificate".

Apache-based Server

Apache web server is another platform you can use to run your HTTP server, in addition to Http.Sys-based Server. If the Http.sys-based server is the recommended way to build Sparkle HTTP server on Microsoft Windows, Apache is the way to go if you want to run it on Linux (although you can also use it on Windows as well).

Apache support on Sparkle is based on built-in Web Broker technology available in Delphi. You create an Apache module using Web Broker and then deploy the module to an Apache web server using standard procedures. Here are the steps:

1. Create an Apache module using Web Broker

Basically, go to *File* > *New* > *Other*, then *Delphi Projects* > *WebBroker* > *Web Server Application* and then choose "Apache dynamic link module". This will create the default Apache module library.

2. In WebModuleUnit1 unit, add the units `Sparkle.WebBroker.Server` and `Sparkle.WebBroker.Adapter` to the uses clause:

```
uses {...},
  Sparkle.WebBroker.Server,
  Sparkle.WebBroker.Adapter,
  Sparkle.HttpServer.Module; // just for the anonymous module example
```

3. Declare a global TWebBrokerServer instance:

```
var
  Server: TWebBrokerServer;
```

4. Create the instance in the initialization of the unit, and add your desired Sparkle modules (XData, RemoteDB, etc.) to its Dispatcher:

```
initialization
  Server := TWebBrokerServer.Create;
  // add modules you want to use. This example assumes a simple Sparkle module,
  // but you can add your XData, RemoteDB or any other Sparkle module
  Server.Dispatcher.AddModule(TAnonymousServerModule.Create(
    'http://localhost/tms/hello',
    procedure(const C: THttpServerContext)
    begin
      C.Response.StatusCode := 200;
      C.Response.ContentType := 'text/html';
      C.Response.Close(TEncoding.UTF8.GetBytes('<h1>Hello, World!</h1>'));
    end
  ));
finalization
  Server.Free;
end.
```

5. Replace the WebModule1DefaultHandlerAction event handler with the code below to dispatch the requests:

```
procedure TWebModule1.WebModule1DefaultHandlerAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Adapter: IWebBrokerAdapter;
begin
  Adapter := TWebBrokerAdapter.Create(Request, Response);
  Server.DispatchRequest(Adapter);
end;
```

That's it, your Apache module is done. You can now just build it and deploy to your Apache server. This is just WebBroker and Apache knowledege, here are some links to videos and/or articles about how to do it:

Building and Deploying a Linux Apache Module with Delphi WebBroker (Youtube video)

Delphi for Linux Database and Web Development (Blog post)

Indy-based Server

Alternatively to Http.Sys-based Server and Apache-based server, TMS Sparkle also offers an Indy-based server. Indy is an HTTP library for Delphi that fully implements HTTP server. While the http.sys-based server is the recommended option for Sparkle HTTP servers on Windows, and Apache is the option for Linux-based servers - both technologies are solid and proven - Indy-based server has its own usage.

With Indy-based server you can have a stand-alone HTTP server in any platform supported by Delphi, so it's an alternative for example to have a Sparkle HTTP server on Android, or full stand-alone HTTP server on Linux (without installing Apache). For Windows it's still recommended using http.sys as you can also build stand-alone server with it.

To use Indy-based server just instantiates the TIndySparkleHTTPServer (declared in unit `Sparkle.Indy.Server`) and use the Dispatcher property which is a Sparkle HTTP Dispatcher for you to register Sparkle modules (like XData, RemoteDB, etc.). Here is a code example:

```
uses
  {...}, Sparkle.Indy.Server;

procedure TForm1.FormCreate(Sender: TObject);
var
  Server: TIndySparkleHTTPServer;
begin
  Server: = TIndySparkleHTTPServer.Create(Self);
  Server.DefaultPort := 8080;
  Server.Dispatcher.AddModule(TAnonymousServerModule.Create(
    'http://localhost:8080/tms/hello',
    procedure(const C: THttpServerContext)
    begin
      C.Response.StatusCode := 200;
      C.Response.ContentType := 'text/html';
      C.Response.Close(TEncoding.UTF8.GetBytes('<h1>Hello, World!</h1>'));
    end
  ));
  Server.Active := True;
end;
```

Some remarks:

- TIndySparkleHttpServer is a TComponent descending from TIdHTTPServer. In the example above it's being created as owned by the Form, so it will be destroyed automatically. You can also simply create with no owner, and then you need to destroy the instance eventually.
- Since it descends from TIdHTTPServer, it inherits all its properties and events. The properties DefaultPort and Active used above, for example, are from TIdHTTPServer. It's out of the scope of this documentation to explain full Indy HTTP server usage.

In-Process Server

Sparkle allows you to create an in-process "HTTP" server. This approach allows you to have a "server" in your application that responds to calls from the client inside the same application. This can serve many purposes:

- **Build a quick prototype** without needing to setup an http.sys-based or Apache-based server.
- **Create a monolithic version** of your application that works standalone, no needing a server, and without changing a single line of code.
- **Test your server modules**.

You can have multiple in-process server in the same application. Each server is identified by a name.

Creating the in-process server

1. The server is created on-the-fly when you try to access it by its name using TInProcHttpServer (declared in unit Sparkle.InProc.Server):

```
uses {...}, Sparkle.InProc.Server;
{...}
var Server: TInProcHttpServer;
{...}
  Server := TInProcHttpServer.Get('myserver');
```

2. Add your modules (XData, RemoteDB, etc.) to the server using the Dispatcher property:

```
Server.Dispatcher.AddModule(SomeModule);
```

That is enough to have the server working. You can even have it in a one-line code:

```
TInProcHttpServer.Get('myserver').Dispatcher.AddModule(SomeModule);
```

Accessing the server from the client

When using Sparkle HTTP Client, or any other class that uses it under the hood, you just need to use an URI with the *local://* prefix, followed by the in-process server name. For example, to perform a request to the server root, just use this:

```
Client.Get('local://myserver');
```

to access another URL in the server, just build it as you would do with regular HTTP requests:

```
Client.Get('local://myserver/somepath?a=1');
```

Note that the server doesn't "run", i.e., it doesn't have a listener or a thread executing to respond to requests. It's the client that detects you are trying to access an in-process server (from the *local://* scheme) and then just directly dispatch the request to the server dispatcher, in the same thread.

HTTP Dispatcher

TMS Sparkle provides four different types of server:

- Http.Sys-based Server
- Apache-based Server
- In-Process Server
- Indy-based Server

All of them have in common the HTTP Dispatcher (THTTPDispatcher class) which is the object that effectively handles the requests. When registering server modules, you register them in the dispatcher (provided by each server type through a property named Dispatcher). All the server does is receive the request and then pass it to the dispatcher.

One of the advantages of using dispatcher is because it's agnostic to the server type. Regardless if the server is http.sys or Apache, for example, you add modules to the dispatcher and don't care about what is the server type. This is also useful to convert your server from one type to another, or even to create an alternative in-process server to test your modules.

For example, you could have a generic method that adds modules to the dispatcher:

```
uses {...}, Sparkle.HttpServer.Dispatcher;

procedure AddModules(Dispatcher: THttpDispatcher);
begin
  Dispatcher.AddModule(TXDataServerModule.Create(XDataUrl, Pool));
  Dispatcher.AddModule(TRemoteDBModule.Create(RemoteDBUrl, Factory));
  Dispatcher.AddModule(TMyCustomModule.Create(CustomUrl));
end;
```

Then you can call that method when creating an http.sys-based server:

```
var
  Server: THttpSysServer;
begin
  Server := THttpSysServer.Create;

  AddModules(Server.Dispatcher);
  Server.Start;

  {...}
```

and you can reuse the same dispatcher setup with an in-process server, for testing purposes for example:

```
AddModules(TInProcHttpServer.Get('myserver').Dispatcher);
```

Server Modules

HTTP Server Modules are the main part of your Sparkle HTTP Server service/application. Those are the objects that will process HTTP requests, and where your logic will be implemented. For more info about how server modules fit into the whole server architecture, see the overview for Sparkle Http Servers.

The basic workflow for creating and using a server module is:

1. Create a new module class inheriting from THttpServerModule class (declared in Sparkle.HttpServer.Module unit);
2. Override the method ProcessRequest and insert code to examine the HTTP request and build the HTTP response;
3. Create an instance of the module passing the base URI which it will respond to;
4. Register the module in the HTTP Dispatcher.

The following example creates a module that responds any request with a "Test123" plain text response.

```
uses {...}, Sparkle.HttpServer.Module, Sparkle.HttpServer.Context;

type
  TSimpleModule = class(THttpServerModule)
    public procedure ProcessRequest(const C: THttpServerContext); override;
  end;

{...} implementation {...}

procedure TSimpleModule.ProcessRequest(const C: THttpServerContext);
begin
  C.Response.StatusCode := 200;
  C.Response.ContentType := 'text/plain';
  C.Response.Close(TEncoding.UTF8.GetBytes('Test123'));
end;

// Add the module to a dispatcher
Dispatcher.AddModule(TSimpleModule.Create('http://host:2001/simple/');
```

The example above is just a simple, unreal example. In real applications, obviously, you will need to carefully examine the HTTP request and then properly build the HTTP response.

TAnonymousServerModule class

For very simple operations, there is TAnonymousServerModule that you can use just by passing an anonymous request procedure to process the request. This way you don't even need to create a new class derived from THttpServerModule. The following code creates and adds a minimal "Hello, World" module with one line of code:

```
Server.AddModule(TAnonymousServerModule.Create(
  'http://+:80/tms/hello',
  procedure(const C: THttpServerContext)
  begin
    C.Response.StatusCode := 200;
    C.Response.ContentType := 'text/html';
    C.Response.Close(TEncoding.UTF8.GetBytes('<h1>Hello, World!</h1>'));
  end
));
```

Examining the Request

When processing a request in a server module, the first step is examine the HTTP request. The only parameter provided by the ProcessRequest method of the server module is a THttpServerContext object. This object has two key properties: Request and Response. The Request property provides a THttpServerRequest object with all the info about the HTTP request sent by the client. THttpServerRequest class is declared in unit `Sparkle.HttpServer.Context`.

A quick example of how to read information from request is showed below. The server module response with a "Method not allowed" error (405) if the client sends a PUT request.

```
procedure TMyServerModule.ProcessRequest(const C: THttpServerContext);
begin
  if C.Request.MethodType = THttpMethod.Put then
    C.Response.StatusCode := 405
  else
    // code for other methods
end;
```

Request URI

You can check the URI requested by the client using property Uri, which returns a TUri object. You can use several properties of TUri object to retrieve information about parts of the URI.

```
RequestedPath := C.Request.Uri.Path;
QueryString := C.Request.Uri.Query;
```

Request Method

Use properties MethodType or Method to retrieve the method in the http request. MethodType property is just an enumerated type with the most commonly used methods to help you with case statements or similar language constructs. The Method property provides you with the textual representation of the method.

```
THttpMethod = (Other, Get, Post, Put, Delete, Head);

{...}

case C.Request.MethodType of
  THttpMethod.Get: DoGetResponse;
  THttpMethod.Put: DoPutResponse;
else
  DoInvalidRequestMethod;
end;
{...}
if C.Request.Method = 'DELETE' then // delete method requested
```

Request Headers

Request headers are available through Headers property of the Request object. You can use methods Exists and Get to check if a header is included in request, or get the header value.

```
Accept := C.Request.Headers.Get('Accept');
```

The Headers property is actually a THttpHeaders object. There are several other methods you can use in this headers object to manipulate headers.

Request body (content)

The request body, or content, is available as a byte array in property Content. If the message has no content, the length of byte array will be zero.

```
ContentAsUTF8String := TEncoding.UTF8.GetString(C.Request.Content);
```

Authenticated user

If the request has been processed by an authentication middleware, it's possible that the user identity and claims are present in the request. To access such info, just read the User property:

```
UserIdentity := C.Request.User;
if UserIdentity <> nil then // means request is authenticated
```

Remote IP

You can check the IP address of the client which sent the request by reading the RemoteIP property.

Building the Response

When processing a request in a server module, you must create the HTTP response. The only parameter provided by the ProcessRequest method of the server module is a THttpServerContext object. This object has two key properties: Request and Response. The Response property provides a THttpServerResponse object with all the properties and methods you need to build the HTTP response.

A quick example is showed below. The server module responses with a "Method not allowed" error (405) if the client sends a PUT request.

```
procedure TMyServerModule.ProcessRequest(const C: THttpServerContext);
begin
  if C.Request.MethodType = THttpMethod.Put then
    C.Response.StatusCode := 405
  else
    // code for other methods
end;
```

Status Code

Use the StatusCode property to specify the HTTP status code response. If you don't set the status code, Sparkle will use the default value, which is 400.

```
C.Response.StatusCode := 200; // OK
```

Response Headers

You can specify the response headers using Headers property of the Response object. The Headers property provides you with a THttpHeaders object. There are several methods you can use in this headers object to manipulate headers. The following example clears headers and set the value of ETag header.

```
C.Response.Headers.Clear;
C.Response.Headers.SetValue('ETag', '"737060cd8c284d8af7ad3082f209582d"');
```

ContentEncoding and ContentType

You can use ContentEncoding and ContentType to set content encoding and type:

```
C.Response.ContentType := 'application/json';
```

Sending response with no content

After you set response headers and specific properties (like status code), you can call Close method to send response headers without content body, and finish the response:

```
C.Response.Close;
```

You don't need to explicitly close the response. If you don't close it, Sparkle will close it later for you. Once a response is closed, it's send back to client, thus you can't change any other property that affect the response anymore (headers, status code, etc.). If you do, an exception will be raised.

Sending content using byte array

You can also close the response sending an array of bytes as content.

```
C.Response.ContentType := 'text/plain';
C.Response.Close(TEncoding.UTF8.GetBytes('A plain text response.'));
```

This will set the content-length header with the correct value, send HTTP response headers, and then send the bytes as the content body of the message.

Sending content using stream

Instead of a byte array, you can use streams to send the content body of response message. You do that by using Content property, which returns a TStream object you can use to write data:

```
MyBitmap.SaveToStream(C.Response.Content);
// You can also use C.Response.Content.Write method
```

When the content stream is written for the first time, Sparkle will send the response headers to the client, and start sending the content body. From this point, the HeadersSent property will be set to true, and any change to any property that affects headers (StatusCode, ContentType, Headers, etc.) will raise an exception. As you keep writing the string, Sparkle will keep sending the bytes to the client. After you finished writing to the stream, call Close method to finish the request. If you don't call Close method, Sparkle will eventually call it at a later time, closing the response.

Setting ContentLength before sending content

If you know the size of the content in advance, you can set ContentLength property before writing to the stream:

```
C.Response.ContentLength := 65200;
```

This has the advance of sending the Content-Length header before sending the content using the stream. If after you set ContentLength property you send the content using byte array by calling Close method (see above), the existing value of ContentLength will be discarded and the length of byte array will be used.

If you send the content body using the Content stream, then the number bytes written to the stream must be equal to the value of ContentLength property. If you write more or less bytes, an exception will be raised.

Sending chunked responses

If you don't know the size of the response in advance, or if you just want to send chunked responsed, you can set Chunked property to true:

```
C.Response.Chunked := true;
// Now write to stream using Content property
```

When you start writing to the stream, headers will be sent without content-length header, and with "transfer-encoding: chunked" header. The content will be sent using chunked encoding, while you keep writing to the stream. You can finish the response by calling Close method. As already mentioned above, Sparkle you later call Close method if you don't.

Checking response status

You can use HeadersSent and Closed property to check the current status of HTTP response message. If no response was sent to client yet, both properties will be false. If the HTTP response message headers were already sent, HeadersSent property will be true and you can't change any property of response that affects headers. If you do, an exception will be raised. If you have called Close method and the response is closed (meaning no more data can be sent to client), Closed property will be true. If you try to send any info to client (e.g., writing to Content stream), an exception will be raised.

```
if not C.Response.HeadersSent then
  C.Response.StatusCode := 200;
if not C.Response.Closed then
  C.Response.Content.Write(Buffer, BufferSize);
```

Handling Multipart Content

Sparkle provides specific classes to handle multipart content sent by the client. More specifically, multipart/form-data content-type. That is usually the content-type sent by browsers when you use the HTML <form> tag to upload a file content (input control with type set to "file").

Key class is TMultipartFormDataReader, declared in unit `Sparkle.Multipart.FormDataReader`. The following is an example of how to use it.

```
uses {...}, Sparkle.Multipart.FormDataReader;

procedure ProcessRequest(const C: THttpServerContext);
var
  Reader: TMultipartFormDataReader;
  I: integer;
  Part: TMultipartFormDataPart;
  vres: string;
begin
  Reader := TMultipartFormDataReader.Create(C.Request.ContentStream, C.Request.He
aders.Get('content-type'));
  try
    vres := '';
    while Reader.Next do
    begin
      Part := Reader.PartInfo;
      vres := vres + 'Name: ' + Part.Name + #13#10;
      if Part.FileName <> '' then
        vres := vres + 'FileName: ' + Part.FileName + #13#10;
      vres := vres + 'MediaType: ' + Part.MediaType + #13#10;
      vres := vres + 'Charset: ' + Part.Charset + #13#10;
      vres := vres + 'Content-Disposition: ' + Part.ContentDisposition + #13#10;

      vres := vres + '=== Params ==='#13#10;

      for I := 0 to Part.ParamCount - 1 do
        vres := vres + Part.ParamName[I] + ': ' + Part.ParamValue[I] + #13#10;

      if (Part.FileName <> '') and (Part.MediaType <> '') then
        TFile.WriteAllBytes(
          (TPath.Combine(ExtractFilePath(ParamStr(0)), TPath.GetFileName(Part.Fil
eName))),
          Reader.ContentAsBytes)
      else
        vres := vres + 'Content: ' + Reader.ContentAsString + #13#10;
      vres := vres + #13#10;
    end;

    C.Response.StatusCode := 200;
    C.Response.ContentType := 'text/plain;charset=UTF-8';
    C.Response.Close(TEncoding.UTF8.GetBytes(vres));
  finally
    Reader.Free;
  end;
end;
```

After creating TMultipartFormDataReader, use Reader.Next to iterate through the parts, and use Reader.PartInfo to get information about that specific part.

Besides the regular properties used in the example above (Name, FileName, MediaType, Charset, ParamCount, ParamName, ParamValue), you can read the part content in three different ways:

- **ContentAsBytes**: Read the content as a byte array TArray<byte>.
- **ContentAsString**: Read the content as string, using the specified charset or UTF8 if not specified.
- **ContentToStream**: Save the part content to a stream. This can be used, for example, to transfer the content to a TFileStream.

It's worth noting that TMultpartFormDataReader is a sequential reader, which means you can start reading the content in chunks. This allows very low memory usage, as you don't need to load all the part content in memory. Using ContentToStream, for example, you can transfer the content sent by the client directly to a file stream keeping the memory usage low.

Design-Time Components

TMS Sparkle provides several components for design-time usage. The main purpose is to provide a RAD experience, by just dropping components in the form and configuring them, allowing setting up servers with almost no line of code.

Even though you can use the components at runtime, creating them from code, that would usually be not necessary, as the components are just **wrappers** for the existing non-component architecture.

As a general rule, you have server components, which inherit from TSparkleServer, that wrap server modules. Each server component must be associated to a dispatcher component (inherited from TSparkleDispatcher), which will handle the HTTP requests.

General usage is:

1. Drop a dispatcher component in the form (for example, TSparkeHttpSysDispatcher);
2. Drop a server component in the form (TSparkleStaticServer, TXDataServer, etc.);
3. Associated the server to the dispatcher through the Dispatcher property of server component;
4. Specify the BaseUrl property of the server (for example, *http://+:2001/tms/myserver*;
5. Configure any specific property of the server;
6. Set Active property of dispatcher component to true.

That would be enough. Run the application and your HTTP server will be running at the specified base url address. You can associated more than one server to the same dispatcher, as long as the servers as configured at different base url addresses.

Server components

A server component wraps a server module. All server components share common features, and below is the list of server components and the associated modules they wrap:

Component	Wrapped module
TSparkleStaticServer	TStaticModule - static file server.
TSparkleGenericServer	TAnonymousServerModule - a generic Sparkle server that handles raw requests.
TXDataServer	TXDataServerModule - the TMS XData server module.

Dispatcher components

A dispatcher component wraps servers like Http.sys-based server or Indy-based server. Here is the list of available dispatcher components:

Component	Wrapped server
TSparkleHttpSysDispatcher	THttpSysServer - It handles HTTP request using Windows http.sys (IIS-based) driver. Provides methods Start, Stop and events OnStart, OnStop.

Server Components - Common Features

Server components wrap Sparkle server modules. There are several Sparkle server components, like TSparkleStaticServer, TSparkleGenericServer or even TXDataServer, but all of them inherit from the base TSparkleServer class and share the common features, available to all of them:

Properties

Name	Description
BaseUrl: string	The root URL where the server will respond requests from. For example: "*http://+:2001/tms/myserver*".
Dispatcher: TSparkleDispatcher	The dispatcher component which will handle the HTTP communication and dispatch the requests to the server.

Events

Name	Description
OnModuleCreate	The event will be fired when the server module instance is created by the component. Since every server component wraps a server module, in the end what the component does it create an instance of the module and pass it to the dispatcher. In this event you will have an opportunity to retrieve the server module instance and change/configure the way you want to. The parameters of the event are Sender (the server component) and Module, which is the server module created.

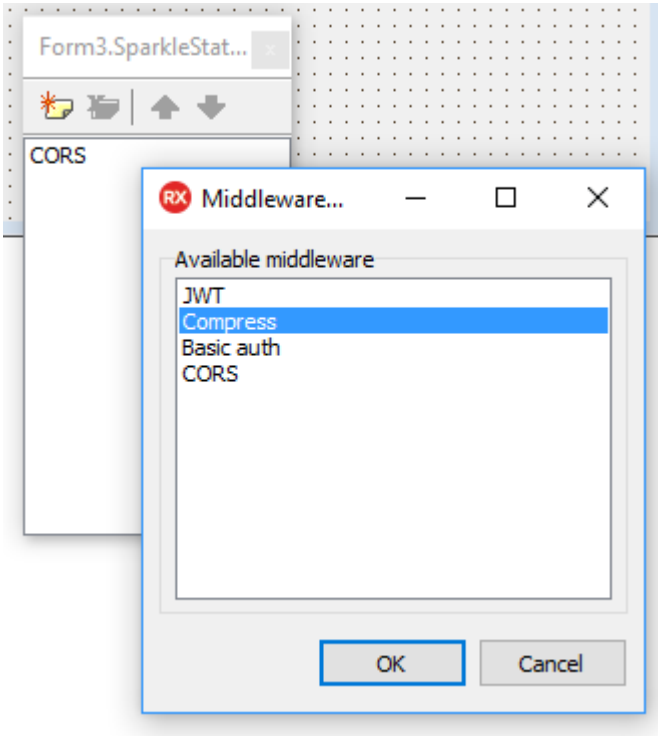
Middleware editor

Any server module can have middleware items added to it. You can directly add middleware to the server module (using the OnModuleCreate event, for example), but the server component already has a design-time middleware editor to make your life easier.

Right-click the component, and choose "Manage middleware list...".

This will open the design-time middleware editor where you can add different types of middleware available (CORS, Compress, JWT, etc.). Select the middleware from the list to see its properties in the object inspector and change it, if necessary.

The design-time middleware items added are just wrappers around the existing middleware classes, you can refer to the available middleware to learn about the properties available and what they are for.



TSparkleStaticServer Component

TSparkleStaticServer component wraps the TStaticModule server module. The purpose is to server static files, as traditional HTTP server.

Besides the features that are common to all server components, TSparkleStaticServer provides the additional properties:

Properties

Name	Description
RootDir: string	Contains the local directory from which the files will be served from. For example: "*C:\mywebfiles*".

TSparkleGenericServer Component

TSparkleGenericServer component wraps the TAnonymousServerModule server module. The purpose is to provide low-level processing to any HTTP request, using an event for that.

Besides the features that are common to all server components, this server provides the additional properties:

Events

Name	Description
OnProcessRequest: TProcessRequestEvent	This event is fired for each incoming HTTP request. Use the request object to read request data, and set the HTTP response using the response object.

Middleware System

TMS Sparkle provides classes that allow you to create middleware interfaces and add them to the request processing pipeline. In other words, you can add custom functionality that pre-process the request before it's effectively processed by the main server module, and post-process the response provided by it.

Using middleware interfaces allows you to easily extend existing server modules without changing them. It also makes the request processing more modular and reusable. For example, the compression middleware is a generic one that compress the server response if the client accepts it. It can be added to any server module (XData, RemoteDB, Echo, or any other module you might create) to add such functionality.

A middleware is represented by the interface IHttpServerMiddleware, declared in unit `Sparkle.HttpServer.Module`. To add a middleware interface to a server module, just use the AddMiddleware function (in this example, we're using a TMS XData module, but can be any Sparkle server module):

```
var
  MyMiddleware: IHttpServerMiddleware;
  Module: TXDataServerModule;
{...}
  // After retrieving the MyMiddleware interface, add it to the dispatcher
  Module.AddMiddleware(MyMiddleware);
  Dispatcher.AddModule(Module);
```

The following topics explain in deeper details how to use middlewares with TMS Sparkle.

Compress Middleware

Use the Compress middleware to allow the server to return the response body in compressed format (gzip or deflate). The compression will happen only if the client sends the request with the header "accept-encoding" including either gzip or deflate encoding. If it does, the server will provide the response body compressed with the requested encoding. If both are accepted, gzip is preferred over deflate.

To use the middleware, just create an instance of TCompressMiddleware (declared in `Sparkle.Middleware.Compress` unit) and add it to the server module:

```
uses {...}, Sparkle.Middleware.Compress;
{...}
  Module.AddMiddleware(TCompressMiddleware.Create);
```

Setting a threshold

TCompressMiddleware provides the Threshold property which allows you to define the minimum size for the response body to be compressed. If it's smaller than that size, no compression will happen, regardless of the 'accept-encoding' request header value. The default value is 1024, but you can change it:

```
var
  Compress: ICompressMiddleware;
begin
  Compress := TCompressMiddleware.Create;
  Compress.Threshold := 2048; // set the threshold as 2k
  Module.AddMiddleware(Compress);
```

CORS Middleware

Use the CORS middleware to add CORS support to the Sparkle module. That will allow web browsers to access your server even if it is in a different domain than the web page.

To use the middleware, just create an instance of TCorsMiddleware (declared in `Sparkle.Middleware.Cors` unit) and add it to the server module:

```
uses {...}, Sparkle.Middleware.Cors;
{...}
  Module.AddMiddleware(TCorsMiddleware.Create);
```

Basically the above code will add the Access-Control-Allow-Origin header to the server response allowing any origin:

```
Access-Control-Allow-Origin: *
```

Additional settings

You can use overloaded Create constructors with additional parameters. First, you can set a different origin for the Access-Control-Allow-Origin:

```
Module.AddMiddleware(TCorsMiddeware.Create('somedomain.com'));
```

You can also configure the HTTP methods allowed by the server, which will be replied in the Access-Control-Allow-Methods header:

```
Module.AddMiddleware(TCorsMiddeware.Create('somedomain.com', 'GET,POST,DELETE,PUT
'));
```

And finally the third parameter will specify the value of Access-Control-Max-Age header:

```
Module.AddMiddleware(TCorsMiddeware.Create('somedomain.com', 'GET,POST,DELETE,PUT
', 1728000));
```

To illustrate, the above line will add the following headers in the response:

```
Access-Control-Allow-Origin: somedomain.com
Access-Control-Allow-Methods: GET,POST,DELETE,PUT
Access-Control-Max-Age: 1728000
```

JWT Authentication Middleware

Add the JWT Authentication Middleware to implement authentication using JSON Web Token. This middleware will process the authorization header, check if there is a JSON Web Token in it, and if it is, create the user identity and claims based on the content of JWT.

The middleware class is TJwtMiddleware, declared in unit `Sparkle.Middleware.Jwt`. It's only available for **Delphi XE6 and up**.

To use the middleware, create it passing the secret to sign the JWT, and then add it to the TXDataServerModule object:

```
uses {...}, Sparkle.Middleware.Jwt;

  Module.AddMiddleware(TJwtMiddleware.Create('my jwt secret'));
```

By default, the middleware rejects expired tokens and allows anonymous access. You can change this behavior by using the Create constructor as explained below in the Methods table.

Properties

Name	Description
Secret: TBytes	The secret used to verify the JWT signature. Usually you won't use this property as the secret is passed in the Create constructor.

Methods

Name	Description
constructor Create(const ASecret: string; AForbidAnonymousAccess: Boolean = False; AAllowExpiredToken: Boolean = False);	Creates the middleware using the secret specified in the ASecret parameter, as string. The string will be converted to bytes using UTF-8 encoding. The second boolean parameter is AForbidAnonymousAccess, which is False by default. That means the middleware will allow requests without a token. Pass True to the second parameter to not allow anonymous access. The third boolean parameter is AAllowExpiredToken which is False by default. That means expired tokens will be rejected by the server. Pass True to the third parameter to allow rejected tokens.
constructor Create(const ASecret: TBytes; AForbidAnonymousAccess: Boolean = False; AAllowExpiredToken: Boolean = False);	Same as above, but creates the middleware using the secret specified in raw bytes instead of string.

Basic Authentication Middleware

Add the Basic Authentication middleware to implement authentication using Basic authentication. This middleware will check the authorization header of the request for user name and password provided using Basic authentication.

The user name and password will then be passed to a callback method that you need to implement to return an user identity and its claims. Different from the JWT Authentication Middleware which automatically creates the identity based on JWT content, in the basic authentication it's up to you to do that, based on user credentials.

If your module implementation returns a status code 401, the middleware will automatically add a www-authenticate header to the response informing the client that the request needs to be authenticated, using the specified realm.

Example:

```
uses {...}, Sparkle.Middleware.BasicAuth, Sparkle.Security;

  Module.AddMiddleware(TBasicAuthMiddleware.Create(
    procedure(const UserName, Password: string; var User: IUserIdentity)
    begin
      // Implement custom logic to authenticate user based on UserName and
Password
      // For example, go to the database, check credentials and then create an
user
      // identity with all permissions (claims) for this user
      User := TUserIdentity.Create;
      User.Claims.AddOrSet('roles').AsString := SomeUserPermission;
      User.Claims.AddOrSet('sub').AsString := UserName;
    end,
    'My Server Realm'
  ));
```

Related types

```
TAuthenticateBasicProc = reference to procedure(const UserName, Password: string;
var User: IUserIdentity);
```

The procedure that will be called for authentication. UserName and Password are the user credentials sent by the client, and you must then create and return the IUserIdentity interface in the User parameter.

Properties

Name	Description
OnAuthenticate: TAuthenticateBasicProc	The authentication callback that will be called when the middleware retrieves the user credentials.
Realm: string	The realm that will be informed in the www-authenticate response header. Default realm is "TMS Sparkle Server".

Methods

Name	Description
constructor Create(AAuthenticateProc: TAuthenticateBasicProc; const ARealm: string)	Creates the middleware using the specified callback and realm value.
constructor Create(AAuthenticateProc: TAuthenticateBasicProc)	Creates the middleware using the specified callback procedure.

Logging Middleware

Add the Logging Middleware to implement log information about requests and responses processed by the Sparkle server. This middleware uses the built-in Logging mechanism to log information about the requests and responses. You must then properly configure the output handlers to specify where data should be logged to.

The middleware class is TLoggingMiddleware, declared in unit `Sparkle.Middleware.Logging`.

To use the middleware, create it, set the desired properties if needed, then add it to the TXDataServerModule object:

```
uses {...}, Sparkle.Middleware.Logging;

var
  LoggingMiddleware: TLoggingMiddleware;

begin
  LoggingMiddleware := TLoggingMiddleware.Create;
  // Set LoggingMiddleware properties.
  Module.AddMiddleware(LoggingMiddleware);
end;
```

You don't need to set any property for the middleware to work. It will output information using the default format.

Properties

Name	Description
FormatString: string	Specifies the format string for the message to be logged for each request/response, according to the format string options, described at the end of this topic. The default format string is: `:method :url :statuscode - :responsetime ms`
ExceptionFormatString: string	Specifies the format string for the message to be logged when an exception happens during the request processing. In other words, if any exception occurs in the server during request processing, and LogExceptions property is True, the exception will be logged using this format. Different from FormatString property which has a specific format, the content of this property is just a simple string used by Delphi Format function. Three parameters are passed to the Format function, in this order: The exception message, the exception class name and the log message itself. Default value is: `(%1:s) %0:s - %2:s`

Name	Description
LogExceptions: Boolean	Indicates if the middleware should catch unhandled exceptions raised by the server and log them. Default is True. Note if LogExceptions is True logged exceptions will not be propagated up in the chain. If LogExceptions is False, the exception will be raised again to be handled by some other middleware or the low level exception handling system in Sparkle.
LogLevel: TLogLevel	The level of log messages generated by the middleware. It could be Trace, Debug, Info, Warning, Error. Default value is Trace.
ExceptionLogLevel: TLogLevel	The level of exception log messages generated by the middleware. It could be Trace, Debug, Info, Warning, Error. Default value is Error.
property OnFilterLog: TLoggingFilterProc	OnFilterLog event is fired for every incoming request that should be logged. This is an opportunity for you to filter which requests should be logged. For example, you might want to only log messages for requests that return an HTTP status code indicating an error. Below is an example of use.
property OnFilterLogEx: TLoggingFilterExProc	OnFilterExLog event is fired for every incoming request that should be logged. This is an opportunity for you to filter which requests should be logged. For example, you might want to only log messages for requests that return an HTTP status code indicating an error. Below is an example of use.

TLoggingFilterProc

```
TLoggingFilterProc = reference to procedure(Context: THttpServerContext; var Acce
pt: Boolean);

LoggingMiddleware.OnFilterLog := procedure(Context: THttpServerContext; var Accep
t: Boolean)
  begin
    Accept := Context.Response.StatusCode >= 400;
  end;
```

TLoggingFilterExProc

```
TLoggingFilterExProc = reference to procedure(Context: THttpServerContext;  Info:
TLoggingInfo; var Accept: Boolean);

LoggingMiddleware.OnFilterLog := procedure(Context: THttpServerContext;
  Info: TLoggingInfo; var Accept: Boolean)
  begin
    if Context.Response.StatusCode >= 400 then
    begin
      // log the messages ourselves, with different level
      Accept := False;
      Info.Logger.Warning(Info.LogMessage);
    end;
  end;
```

Format string options

The format string is a string that represents a single log line and utilize a token syntax. Tokens are references by :token-name. If tokens accept arguments, they can be passed using [], for example: :token-name[param] would pass the string 'param' as an argument to the token token-name.

Each token in the format string will be replaced by the actual content. As an example, the default format string for the logging middleware is

```
:method :url :statuscode - :responsetime ms
```

which means it will output the HTTP method of the request, the request URL, the status code, an hifen, then the response time followed by the letters "ms". This is an example of such output:

```
GET /request/path 200 - 4.12 ms
```

Here is the list of available tokens:

:method
The HTTP method of the request, e.g. "GET" or "POST".

:protocol The HTTP protocol of the request, e.g. "HTTP1/1".

:req[header]
The given header of the request. If the header is not present, the value will be displayed as an empty string in the log. Example: :req[content-type] might output "text/plain".

:reqheaders
All the request headers in raw format.

:remoteaddr
The remote address of the request.

:res[header]
The given header of the response. If the header is not present, the value will be displayed as an empty string in the log. Example: :res[content-type] might output "text/plain".

:resheaders
All the response headers in raw format.

:responsetime[digits]
The time between the request coming into the logging middleware and when the response headers are written, in milliseconds. The digits argument is a number that specifies the number of digits to include on the number, defaulting to 2.

:statuscode
The status code of the response.

:url
The URL of the request.

Encryption Middleware

Add the Encryption Middleware to allow for custom encryption of request and response body message. With this middleware you can add custom functions that process the body content, receiving the input bytes and returning processed output bytes.

The middleware class is TEncryptionMiddleware, declared in unit `Sparkle.Encryption.Logging`.

To use the middleware, create it, set the desired properties if needed, then add it to the TXDataServerModule object:

```
uses {...}, Sparkle.Middleware.Encryption;

var
  EncryptionMiddleware: TEncryptionMiddleware;

begin
  EncryptionMiddleware := TEncryptionMiddleware.Create;
  // Set EncryptionMiddleware properties.
  Module.AddMiddleware(EncryptionMiddleware);
end;
```

These are the properties you can set. You need to set at the very minimum either EncryptBytes and DecryptBytes properties.

Properties

Name	Description
CustomHeader: string	If different from empty string, then the request and response will not be performed if the request from the client includes HTTP header with the same name as CustomHeader property and same value as CustomHeaderValue property.
CustomHeaderValue: string	See CustomHeader property for more information.

Name	Description
EncryptBytes: TEncryptDecryptBytesFunc	Set EncryptBytes function to define the encryption processing of the message body. This function is called for every message body that needs to be encrypted. The bytes to be encrypted are passed in the ASource parameter, and the processed bytes should be provided in the ADest parameter. If the function returns True, then the encrypted message (content of ADest) will be used. If the function returns False, then the content of ASource will be used (thus the original message will be used).
DecryptBytes: TEncryptDecryptBytesFunc	Set the DecryptBytes function to define the decryption processing of the message body. See property EncryptBytes for more details.

TEncryptDecryptBytesFunc

```
TEncryptDecryptBytesFunc = reference to function(const ASource: TBytes; out
ADest: TBytes): Boolean;
```

Creating Custom Middleware

To create a new middleware, create a new class descending from THttpServerMiddleware (declared in `Sparkle.HttpServer.Module`) and override the method ProcessRequest:

```
uses {...}, Sparkle.HttpServer.Module;

type
  TMyMiddleware = class(THttpServerMiddleware, IHttpServerMiddleware)
  protected
    procedure ProcessRequest(Context: THttpServerContext; Next: THttpServerProc);
override;
  end;
```

In the ProcessRequest method you do any processing of the request you want, and then you should call the Next function to pass the request to the next process requestor in the pipeline (until it reaches the main server module processor):

```
procedure TMyMiddleware.ProcessRequest(Context: THttpServerContext; Next: THttpSe
rverProc);
begin
  if Context.Request.Headers.Exists('custom-header') then
    Next(Context)
  else
    Context.Response.StatusCode := 500;
end;
```

In the example above, the middleware checks if the header "custom-header" is present in the request. If it does, then it calls the next processor which will continue to process the request normally. If it does not, a 500 status code is returned and processing is done. You can of course modify the request object before forwarding it to the next processor. Then you can use this middleware just by adding it to any server module:

```
Module.AddMiddleware(TMyMiddleware.Create);
```

Alternatively, you can use the TAnonymousMiddleware (unit `Sparkle.HttpServer.Module`) to quickly create a simple middleware without needing to create a new class. The following example does the same as above:

```
Module.AddMiddleware(TAnonymousMiddleware.Create(
procedure(Context: THttpServerContext; Next: THttpServerProc)
begin
  if Context.Request.Headers.Exists('custom-header') then
    Next(Context)
  else
    Context.Response.StatusCode := 500;
end
));
```

Processing the response

Processing the response requires a different approach because the request must reach the final processor until the response can be post-processed by the middleware. To do that, you need to use the OnHeaders method of the response object. This method is called right after the response headers are build by the final processor, but right before it's sent to the client. So the middleware has the opportunity to get the final response but still modify it before it's sent:

```
procedure TMyMiddleware2.ProcessRequest(C: THttpServerContext; Next: THttpServerP
roc);
begin
  C.Response.OnHeaders(
    procedure(Resp: THttpServerResponse)
    begin
      if Resp.Headers.Exists('some-header') then
        Resp.StatusCode := 500;
    end
  );
  Next(C);
end;
```

The above middleware code means: when the response is about to be sent to the client, check if the response has the header "some-header". If it does, then return with status code of 500. Otherwise, continue normally.

Generic Middleware

Generic middleware provides you an opportunity to add code that does any custom processing for the middleware that is not covered by the existing pre-made middleware classes.

The middleware class is TSparkleGenericMiddleware, declared in unit `Sparkle.Comp.GenericMiddleware`. It's actually just a wrapper around the techniques described to create a custom middleware, but available at design-time.

This middleware publishes two events:

Events

Name	Description
OnRequest: TMiddlewareRequestEvent	This event is fired to execute the custom code for the middleware. Please refer to Creating Custom Middleware to know how to implement such code. If you don't call at least Next(Context) in the code, your server will not work as the request chain will not be forwarded to your module. Below is an example.
OnMiddlewareCreate: TMiddlewareCreateEvent	Fired when the IHttpServerMiddleware interface is created. You can use this event to actually create your own IHttpServerMiddleware interface and pass it in the *Middleware* parameter to be used by the Sparkle module.

TMiddlewareRequestEvent

```
TMiddlewareRequestEvent = procedure(Sender: TObject; Context: THttpServerContext;
Next: THttpServerProc) of object;

procedure TForm3.XDataServer1GenericRequest(Sender: TObject;
  Context: THttpServerContext; Next: THttpServerProc);
begin
  if Context.Request.Headers.Exists('custom-header') then
    Next(Context)
  else
    Context.Response.StatusCode := 500;
end;
```

TMiddlewareCreateEvent

```
TMiddlewareCreateEvent = procedure(Sender: TObject; var Middleware: IHttpServerMi
ddleware) of object;
```

Authentication and Authorization

TMS Sparkle provides easy-to-use, built-in authentication and authorization mechanisms. You can use several authentication mechanisms, like Basic or JWT, and implement an agnostic authorization server mechanism, independent of the authentication used.

Here are the basic steps to implement it:

1. Add one of more authentication middleware to your sparkle module. For example, Basic Authenticaction Middleware or JWT Authentication Middleware.
2. Depending on the middleware used, upon authentication create the user identity and its claims.
3. When implementing your server, authorize your requests, by checking for the User property in the request, verify if it exists (authenticated) and if it has the claims needed to perform the server request (authorization).

The following topics explain the above steps in details and provide additional info.

Adding Authentication Middleware

To authenticate your incoming request, you need to add an authentication middleware to your TMS Sparkle module. The purpose of the authentication middleware is check for user credentials sent by the client, and then creating the user identity and claims based on those credentials. The user identity/claims will be added to the request, which will then be forwarded to the next request processor in the middleware pipeline. From that point, you will be able to use that info to authorize your requests.

These are the available authentication middleware and a simple example about how to use them. For more info, go to the specific authentication middleware topic.

JWT (JSON Web Token) Middleware (Delphi XE6 and up only)

Authenticates your requests using JWT (JSON Web Token). This will look for a Bearer token in the request authentication header which contains the JWT. The user identity and claims will be automatically retrieved from the JWT itself. Basic usage is just create the middleware with the secret used to sign the token:

```
uses {...}, Sparkle.Middleware.Jwt;

  Module.AddMiddleware(TJwtMiddleware.Create('my jwt secret'));
```

The middleware just validates and extracts the token information. To proper create a full JWT authentication mechanism, your server has to somehow generate a token for the client (for example, upon a Login method). This is explained in more details in the Authentication Example using JWT (JSON Web Token).

Basic Authentication Middleware

Authenticates requests using Basic Authentication. It looks for the Basic keyword in the request authentication header and retrieves user name and password. It successful it pass user name and password to an event handler that should in turn create and return the proper user identity and claims. Here is an example:

```
uses {...}, Sparkle.Middleware.BasicAuth, Sparkle.Security;

  Module.AddMiddleware(TBasicAuthMiddleware.Create(
    procedure(const UserName, Password: string; var User: IUserIdentity)
    begin
      // Implement custom logic to authenticate user based on UserName and
Password
      // For example, go to the database, check credentials and then create an
user
      // identity with all permissions (claims) for this user
      User := TUserIdentity.Create;
      User.Claims.AddOrSet('roles').AsString := SomeUserPermission;
      User.Claims.AddOrSet('sub').AsString := UserName;
    end,
    'My Server Realm'
  ));
```

User Identity and Claims

The authentication/authorization mechanism is based on user identity and claims. That is represented in Sparkle by the IUserIdentity interface, declared in unit `User.Security`:

```
IUserIdentity = interface
  function Claims: TUserClaims;
end;
```

Such interface has a single function which return the user claims. Claims is a name/value mapping that holds information about the user, like its name, e-mail address, permissions, and any extra info you can add it. Each claim has a name and a value, and the value can be of several types, like integer, string, etc.

The user identity is created and filled by the authentication middleware, and attached to the THttpRequest object in the User property.

If you are implementing the authentication middleware processing, you might want to create a new user identity (class TUserIdentity is a ready-to-use class that implements IUserIdentity interface) and add claims to it using AddOrSet method, for example:

```
uses {...}, Sparkle.Middleware.BasicAuth, Sparkle.Security;

  Module.AddMiddleware(TBasicAuthMiddleware.Create(
    procedure(const UserName, Password: string; var User: IUserIdentity)
    begin
      // Implement custom logic to authenticate user based on UserName and
Password
      // For example, go to the database, check credentials and then create an
user
      // identity with all permissions (claims) for this user
      User := TUserIdentity.Create;
      User.Claims.AddOrSet('roles').AsString := SomeUserPermission;
      User.Claims.AddOrSet('sub').AsString := UserName;
    end,
    'My Server Realm'
  ));
```

If you are implementing your server logic and wants to check user identity and claims, you can use methods Exists or Find to get the user claim and then its value:

```
var RolesClaim: TUserClaim;
begin
  RolesClaim := nil;
  if Request.User <> nil then
    RolesClaim := Request.User.Find('roles');
  if RolesClaim <> nil then
    Permissions := RolesClaim.AsString;
```

TUserClaims holds a list of TUserClaim objects which are destroyed when TUserClaims is destroyed. TUserClaims and TUserClaim classes provide several properties and methods for you to create and read claims.

TUserClaims Methods

Name	Description
function AddOrSet(Claim: TUserClaim): TUserClaim	Adds a new claim to the user. If there is already a claim with the same name as the one being passed, it will be replaced by the new one. You don't need to destroy the TUserClaim object, it will be destroyed when TUserClaims is destroyed.
function AddOrSet(const Name: string): TUserClaim	Creates and adds a new claim with the specified name. If there is already a claim with same name, it will be destroyed and replaced by the new one. You don't need to destroy the new claim. Example: `User.Claims.AddOrSet('isadmin').AsBoolean := True;`

Name	Description
function AddOrSet(const Name, Value: string): TUserClaim	Creates and adds a new claim with the specified name and string value. If there is already a claim with same name, it will be destroyed and replaced by the new one. You don't need to destroy the new claim. Example: `User.Claims.AddOrSet('email', 'user@myserver.com');`
function Exists(const Name: string): Boolean	Returns true if a claim with specified name exists.
function Find(const Name: string): TUserClaim	Returns the TUserClaim object with the specified name. If it doesn't exist, returns nil.
procedure Remove(const Name: string)	Removes and destroys the claim with the specified name.
property Items[const Name: string]: TUserClaim	Returns a claim with the specified name. If the claim doesn't exist, an error is raised.
for Claim in User.Claims	You can use the `for..in` operator to enumerate all the claims in the user identity.

TUserClaim Properties

Name	Description
Name: string	Contains the name of the claim.
AsString: string	Reads or writes the claim value as a string. If the value was not previously saved as a string, an error may be raised.
AsInteger: Int64	Reads or writes the claim value as an Int64 value. If the value was not previously saved as an Int64 value, an error may be raised.
AsDouble: Double	Reads or writes the claim value as a double value. If the value is not a double value, an error may be raised.
AsBoolean: Boolean	Reads or writes the claim value as a boolean value. If the value was not previously saved as a boolean value, an error may be raised.
AsEpoch: TDateTime	Reads or writes the claim value as a date time value. When writing, saves it as a Unix time. When reading, it considers the existing value is a valid Unix time, and converts it to TDateTime. If the value is not a valid Unix time, an exception is raised.

Authorizing Requests

If you have property added an authentication middleware, the HTTP request you receive will already contain authentication information. Implementing your code is just as simple of examining the User property of the request, check if it exists and check the claims it contains.

Suppose a very simple "hello world" module that refuses anonymous connections and only respond to requests from users which has a claim 'isadmin' set to true. It also considers that the authentication mechanism contains a claim named 'sub' which has the name of the user authenticated.

```
uses {...}, Sparkle.Security;

procedure TProtectedHelloModule.ProcessRequest(const C: THttpServerContext);
var
  User: IUserIdentity;
  AdminClaim, NameClaim: TUserClaim;
  ResponseText: string;
begin
  User := C.Request.User;
  if User = nil then
    // not authenticated
    C.Response.StatusCode := 401
  else
  begin
    AdminClaim := User.Claims.Find('isadmin');
    if not (Assigned(AdminClaim) and AdminClaim.AsBoolean) then
      // forbidden
      C.Response.StatusCode := 403;
    else
    begin
      NameClaim := User.Claims.Find('sub');
      ResponseText := 'Hello';
      if NameClaim <> nil then
        ResponseText := ResponseText + ', ' + NameClaim.AsString;


      C.Response.StatusCode := 200;
      C.Response.ContentType := 'text/plain';
      C.Response.Close(TEncoding.UTF8.GetBytes(ResponseText));
    end;
  end;
end;
```

A more detailed example in Authentication Example using JWT (JSON Web Token).

Creating JSON Web Tokens

For JWT handling, Sparkle users a slightly modified version of the nice open source Delphi JOSE and JWT Library: http://github.com/paolo-rossi/delphi-jose-jwt.

You can refer to that library page and documentation to learn how to create the tokens, for example to implement a login mechanism. The library provided in the Sparkle is mostly the same, with the main different that all units names are prefixed with "Bcl." to avoid unit name conflict. So for example, the unit `JSON.Core.Builder` becomes `Bcl.Json.Core.Builder`.

Just for a reference, here is an example about how to generate a JWT:

```
function TTestService.Login(const UserName, Password: string): string;
var
  JWT: TJWT;
  PermissionsFromDatabase: string;
begin
  // check if UserName and Password are valid, and retrieve User data from
database
  // for example, PermissionsFromDatabase, and set desired claims accordingly
  JWT := TJWT.Create(TJWTClaims);
  try
    JWT.Claims.SetClaimOfType<string>('roles', PermissionsFromDatabase);
    JWT.Claims.SetClaimOfType<string>('roles', UserName);
    JWT.Claims.Issuer := 'XData Server';

    JWT.Claims.Expiration := IncHour(Now, 1); // Expire token one hour from now
    Result := TJOSE.SHA256CompactToken('my JWT secret', JWT);
  finally
    JWT.Free;
  end;
end;
```

Built-in Modules

Sparkle is based in a server module system. You can build your own Sparkle modules by processing the requests manually. There are also big Sparkle module implementations which are complex enough to be separate frameworks themselves, like TMS RemoteDB and TMS XData.

And in TMS Sparkle itself there are some modules that are available and ready-to-use for several purposes. Below you will find the list of available Sparkle modules.

TStaticModule

You can use TStaticModule to serve static files from a Sparkle server, pretty much like a regular static web server. It's declared in unit `Sparkle.Module.Static`.

The following example will serve all files under directory "C:\myfiles" at the address "http://<server>:2001/tms/files".

```
uses
  {...}, Sparkle.Module.Static;

var
  Module: TStaticModule;
begin
  Module := TStaticModule.Create('http://+:2001/tms/files', 'C:\myfiles');
  Server.AddModule(Module);
end;
```

Constructors

Name	Description
constructor Create(const ABaseUri: string)	Creates the static module that handle requests for the specified ABaseUri address.
constructor Create(const ABaseUri, ARootDir: string)	Creates the static module that handle requests for the specified ABaseUri address, and servers files from the ARootDir directory.

Properties

Name	Description
RootDir: string	The base local directory where files will be served from.

Name	Description
IndexFiles: TStrings	A list of files names that would be displayed (if present) when the requested address is the directory itself. By default, this property includes "index.html" and "index.htm". This means that whenever the client requests an url with no file name (for example, "http://localhost:2001/tms/files" or "http://localhost:2001/tms/files/subdir", the module will look for any file with the same name as any entry in IndexFiels property. If found, the file will be returned to the client. If no file is found or IndexFiles is empty, a not found (404) error will be returned.
LastModified: Boolean	If True, server will add a Last-Modifed header in response with the date/time when the provided file was last modified, locally. Can be used by clients and browsers for cached responses. It's true by default.
property FolderUrlMode: TFolderUrlMode	Indicates how static module should treat requests to URL addresses (that represent folders, not files) with trailing slash or without trailing slash (for example, "<server>/foldername" and "<server>/foldername/". See below the options for this property.

TFolderUrlMode

- *TFolderUrlMode.RedirectToSlash*: All requests without a trailing slash in URL will 301 redirect to the URL with trailing slash. This is default behavior.
- *TFolderUrlMode.RedirectFromSlash*: All requests with trailing slash in URL will 301 redirect to the URL without trailing slash.
- *TFolderUrlMode.NoRedirect*: No redirect will be performed.

TAnonymousServerModule

You can use TAnonymousServerModule to perform raw, low-level HTTP request processing, in a direct way without having to inherit a new class from THttpServerModule and overriding ProcessRequest method. It's declared in unit `Sparkle.HttpServer.Module`.

The following example will respond with HTTP status code 200 to all requests and send back the same content received.

```
uses
  {...}, Sparkle.HttpServer.Module;

begin
  Server.AddModule(TAnonymousServerModule.Create(
    procedure(const C: THttpServerContext)
    begin
      C.Response.StatusCode := 200;
      C.Response.ContentType := C.Request.Headers.Get('content-type');
      C.Response.ContentLength := Length(C.Request.Content);
      C.Response.Content.Write(C.Request.Content[0], C.Response.ContentLength);
    end;
  ));
end;
```

Constructors

Name	Description
constructor Create(const ABaseUri, AProc: TRequestProc)	Creates the static module that handle requests for the specified ABaseUri address, and servers files from the ARootDir directory.

JSON Classes

Sparkle provides classes for manipulation JSON representation of data. When receiving or sending client requests and server responses, you can use classes available by Sparkle to easily create or read data in JSON. The following topics provide details about the available classes in Sparkle for JSON manipulation.

JSON Writer

Sparkle provides the TJsonWriter class (declared in unit `Sparkle.Json.Writer.pas`) that allows you to generate JSON-encoded data to a stream. To use it, just create an instance of TJsonWriter class passing a TStream object to its constructor method, and then use the methods available in the class to start encoding data in JSON format. Data will be saved to the stream.

TJsonWriter class is written to be a lightweight, fast, unidirectional way to generate JSON data.

Creating the writer

```
JsonWriter := TJsonWriter.Create(MyOutputStream);
```

Available methods

```
function WriteBeginArray: TJsonWriter;
function WriteEndArray: TJsonWriter;
```

Respectively begins a new JSON array, then ends the current array.

```
function WriteBeginObject: TJsonWriter;
function WriteEndObject: TJsonWriter;
```

Respectively begins a new JSON object, then ends the current object.

```
function WriteName(const Name: string): TJsonWriter;
```

Writes the property name of an JSON object.

```
function WriteString(const Value: string): TJsonWriter;
```

Writes a JSON string value. Value will be escaped, if needed.

```
function WriteRawString(const Value: string): TJsonWriter;
```

Writers a JSON string value, without escaping. The exact content of Value parameter will be output as a JSON string, so you must be sure escaping is correct.

```
function WriteBoolean(const Value: boolean): TJsonWriter;
```

Writes a JSON true or false value.

```
function WriteNull: TJsonWriter;
```

Writes a JSON null value.

```
function WriteDouble(const Value: double): TJsonWriter;
```

Writes a double value as a JSON number.

```
function WriteInteger(const Value: Int64): TJsonWriter;
```

Writes an integer value as a JSON number.

```
procedure Flush;
```

Force pending data to be saved in stream.

```
procedure Close;
```

Force pending data to be saved in stream and check for inconsistencies in output (for example, an array that was not closed).

Available properties

```
property IndentLength: integer;
```

Specifies the length of indentation in characters for the output. Default value is zero which means no indentation or line breaks are added to the output stream, generating a compact representation. If IndentLength is greater then zero, then the output will be indented for a better visual representation of the output.

Remarks

You don't need to call Close method, you can use it just to be sure the JSON representation is correct. You also don't need to call Flush explicitly, if you destroy the writer, all pending data will be flushed automatically. But be aware that data is not written directly to the output stream. An internal cache is used for performance reasons, thus if you need to be sure all written data is in the output stream, always call Flush method for that.

All write methods return the instance of TJsonWriter object, which allows you to use methods using fluent interface, for example:

```
JsonWriter
  .WriteBeginObject
  .WriteName('test')
  .WriteInteger(12)
  .WriteEndObject;
```

While writing to the stream, the writer might raise errors if the method is called in the incorrect context. For example, if you try ending an array without previously opening an array, or trying to write a property name without previously started an object.

Example

Give the following string with JSON representation:

```
{
    "name": "John",
    "count": 562,
    "items": [
        "one",
        true,
        null
    ],
    "emptyobject": {},
    "object": {
        "total": 3.14E-10,
        "emptyarray": []
     }
}
```

This is the code used to generate such JSON in the stream specified by AStream variable:

```
Writer := TJsonWriter.Create(AStream);
Writer
  .WriteBeginObject
    .WriteName('name')
    .WriteString('John')
    .WriteName('count')
    .WriteInteger(562)
    .WriteName('items')
    .WriteBeginArray
      .WriteString('one')
      .WriteBoolean(true)
      .WriteNull
    .WriteEndArray
    .WriteName('emptyobject')
    .WriteBeginObject
    .WriteEndObject
```

```
    .WriteName('object')
    .WriteBeginObject
      .WriteName('total')
      .WriteDouble(3.14e-10)
      .WriteName('emptyarray')
      .WriteBeginArray
      .WriteEndArray
    .WriteEndObject
  .WriteEndObject;
Writer.Free;
```

JSON Reader

Sparkle provides the TJsonReader class (declared in unit `Sparkle.Json.Reader.pas`) that allows you to read tokens and values from a stream containing JSON-encoded data. To use it, just create an instance of TJsonReader class passing a TStream object to its constructor method, and then use the methods available in the class to start extracting information from it.

TJsonReader class is written to be a lightweight, fast, unidirectional way to read JSON tokens and values. The tokens are traversed in depth-first order, the same order they appear in the stream.

Creating the reader

```
JsonReader := TJsonReader.Create(MyInputStream);
```

TJsonToken enumerated type

```
type
  TJsonToken = (BeginObject, EndObject, BeginArray, EndArray, Name, Boolean,
Null, Text, Number, EOF);
```

Contains all the possible tokens that are extracted by the reader:

- *BeginObject* and *EndObject* indicate the begin or end of an object.
- *BeginArray* and *EndArray* indicate the begin or end of an array.
- *Name* indicates it's the property name (in a name/value pair) of an object.
- *Boolean* and *Null* refer to JSON boolean (true/false) and JSON null values.
- *Text* refers to a JSON string.
- *Number* refers to a JSON number.
- *EOF* indicates the JSON representation has finished.

Available methods

```
procedure ReadBeginArray;
procedure ReadEndArray;
```

Consumes the next token from the stream, ensuring it's BeginArray or EndArray, respectively.

```
procedure ReadBeginObject;
procedure ReadEndObject;
```

Consumes the next token from the stream, ensuring it's BeginObject or EndObject, respectively.

```
function ReadName: string;
```

Ensures next token is TJsonToken.Name (a property name), consumes it and return its content.

```
function ReadString: string;
```

Read the next value as string, consuming the token. If next token is a number, the value will be returned as a string and token will be consumed. If next token is neither Number (JSON number) nor Text (JSON string) an error will be raised. The string return is already unescaped.

```
function ReadBoolean: boolean;
```

Ensures next token is TJsonToken.Boolean, consumes it and returns its content as boolean value.

```
function ReadDouble: double;
```

Read the next value as double, consuming the token. If next token is a string containing a number representation, the value will be returned as double and token will be consumed. If next token is neither Number (JSON number) or Text (JSON string), nor if next token is Text but contains a value that cannot be converted to a double value, an error will be raised.

```
function ReadInt64: Int64;
function ReadInteger: integer;
```

Read the next value as integer (or Int64), consuming the token. If next token is a string containing a number representation, the value will be returned as integer (or Int64) and token will be consumed. If next token is neither Number (JSON number) nor Text (JSON string), or if next token is Text but contains a value that cannot be converted to an integer value, an error will be raised.

```
procedure SkipValue;
```

Skips the current value and advances to next token. If the current value is a JSON object or JSON array, it will skip the whole object and array until the token right after the object or array.

```
procedure ReadNull;
```

Ensures next token is TJsonToken.Null and consumes it.

```
function HasNext: boolean;
```

Returns true if the current array or object has another element (if the array contains another value, or if object contains another name/value pair).

```
function Peek: TJsonToken;
```

Returns the type of next token in the stream, without consuming it.

```
function IsNull: boolean;
```

Indicates if next token is TJsonToken.Null.

```
function IsFloat: boolean;
```

Indicates if next token is a floating point number. This is used to differentiate the number types. If the number fits is an integer (32 or 64 bit), this function will return false.

```
function IsInt64: boolean;
```

Indicates if next token is an Int64 type. This is used to differentiate the number types. If the number fits in a 32-bit integer, this function will return false.

```
function Eof: boolean;
```

Indicates if next token is TJsonToken.EOF.

NOTE

All methods might raise errors if they try to consume a token that is not the next token. For example, calling ReadBeginArray will raise an error if the next token is not an array begin ("[" character), at the same time ReadInteger raises an error if next token is not a JSON number with a valid integer value.

Examples

Give the following string with JSON representation:

```
{
    "name": "John",
    "count": 562,
    "items": [
        "one",
        true,
        null
    ],
    "emptyobject": {},
    "object": {
        "total": 3.14E-10,
        "emptyarray": []
     }
}
```

This is the code used to read such JSON from the stream specified by AStream variable:

```
Reader := TJsonReader.Create(AStream);

Reader.ReadBeginObject;
  Name := Reader.ReadName; // "name"
  StrValue := Reader.ReadString; // "John"

  Name := Reader.ReadName; // "count"
  IntValue := Reader.ReadInteger; // 562

  Name := Reader.ReadName; // "items"
  Reader.ReadBeginArray;
    StrValue := Reader.ReadString; // "one"
    BoolValue := Reader.ReadBoolean; // true
    Reader.ReadNull; // null
  Reader.ReadEndArray;

  Name := Reader.ReadName; // "emptyobject"
  Reader.ReadBeginObject;
  Reader.ReadEndObject;

  Name := Reader.ReadName; // "object"
  Reader.ReadBeginObject;
    Name := Reader.ReadName; // "total"
    DoubleValue := Reader.ReadDouble; // 3.14E-10

    Name := Reader.ReadName; // "emptyarray"
    Reader.ReadBeginArray;
    Reader.ReadEndArray;
  Reader.ReadEndObject;
Reader.ReadEndObject;

Reader.Free;
```

The following example shows how to use HasNext function to iterate through array values or object properties, and how to use Peek to know what the type of next token and perform actions based on that.

```
Reader.ReadBeginObject;
while Reader.HasNext do
begin
  PropertyName := Reader.ReadName;
  case Reader.Peek of
    TJsonToken.Text: StrValue := Reader.ReadString;
    TJsonToken.Number: DoubleValue := Reader.ReadNumber;
    TJsonToken.Boolean: BooleanValue := Reader.ReadBoolean;
    TJsonToken.Null: Reader.ReadNull;
    TJsonToken.BeginObject: ProcessObject;
    TJsonToken.BeginArray: ProcessArray;
  end;
end;
Reader.ReadEndObject;
```

Logging

TMS Business includes a logging system to help you log data. It's specially useful for server applications, but you can use it anywhere you want. Writing log messages is clean, simple and straightforward. When logging messages, you don't care what the output will be.

```
uses {...}, Bcl.Logging;
var
  Logger: ILogger;
begin
  Logger := LogManager.GetLogger;
  Logger.Error('This is an error!!!');
end;
```

At the beginning of the application, you simply configure the output, which can use the simple built-in debugger, or use the full TMS Logging framework, which can output to several places like Text files, CSV file, TCP/IP listener, Browser, Windows Event Log, among others.

You can also use the Logging Middleware to automatically log HTTP requests and responses processed by Sparkle servers.

Related topics:

- Writing Log Messages
- Log Output Handlers and TMS Logging
- Logging Middleware

Writing Log Messages

Writing log messages (generating log information) with TMS Business is very straightforward. You just need to retrieve the ILogger interface (using LogManager.GetLogger method) and use one of its methods:

```
uses {...}, Bcl.Logging;

var
  Logger: ILogger;
begin
  Logger := LogManager.GetLogger('MyLogger');
  Logger.Error('This is an error!!!');
  Logger.Warning('Pay attention to this');
  Logger.Info('Relevant information');
  Logger.Debug('Some debug output');
  Logger.Trace('Tracing information');
end;
```

Naming loggers

When retrieving loggers using GetLogger, you can give names to your logger. In the example above, the logger will be called MyLogger. You can simply call GetLogger without any name:

```
Logger := LogManager.GetLogger;
```

Which creates a logger with empty name. You can also pass a TClass parameter:

```
Logger := LogManager.GetLogger(TMyClass);
```

It will create a logger which name is the fully qualified name of the class (for example, *MyUnit.TMyClass*).

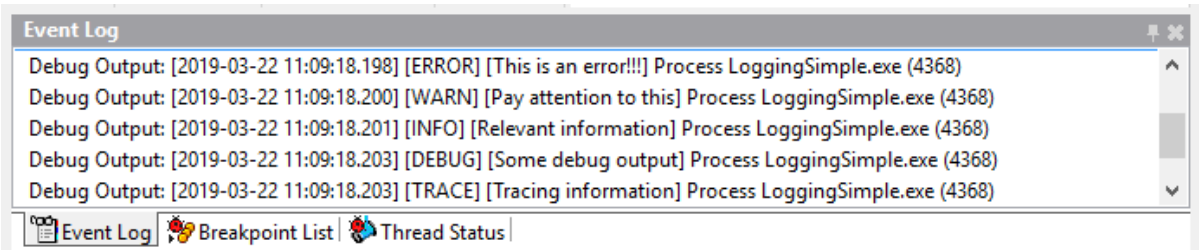
Naming loggers is useful to properly redirect log messages to different outputs. By using names you can redirect messages from specific loggers to one output, ignore messages from other loggers, etc.. More info in topic Log Output Handlers and TMS Logging.

Output

That's pretty much about it for logging data. You don't worry about where the data will be logged to, just log information from where you need. By default, messages won't be output to anywhere. You can quickly see your log messages by calling RegisterDebugLogger anywhere in your application before you start logging:

```
RegisterDebugLogger;
```

This will enable a basic logger that outputs data to the default debugger. You can see log messages in Output tab of Delphi Messages window:



Of course you can choose to output your log messages to a wide range of other options. That is accomplished by using the integration with TMS Logging framework, which provides several output handlers you can choose from.

As a final note, here is the full reference for the ILogger interface:

```
ILogger = interface
  procedure Error(const AValue: TValue);
  procedure Warning(const AValue: TValue);
  procedure Info(const AValue: TValue);
  procedure Trace(const AValue: TValue);
  procedure Debug(const AValue: TValue);
end;
```

Note that the methods accept a TValue parameter, meaning you are not restricted to pass a string to it, but also any other value, like boolean, integer, double, or even complex ones like an object. The way such data will be output will depend on the output handler. For example, the basic Debug Logger just calls TValue.ToString method for getting representation of logged data, which doesn't provide much info for objects. But output handlers in TMS Logging handle objects better, by listing its properties. Thus you can be confident data of primitive types will be properly logging, but for complex data that depends on the output handler.

Log Output Handlers and TMS Logging

You can easily integrate TMS Logging to write your log messages to different targets. TMS Logging is a full-featured logging framework that works independently from TMS Sparkle. To plug TMS Logging as the logging framework, use unit `Bcl.TMSLogging` and call RegisterTMSLogger event.

Using global default logger

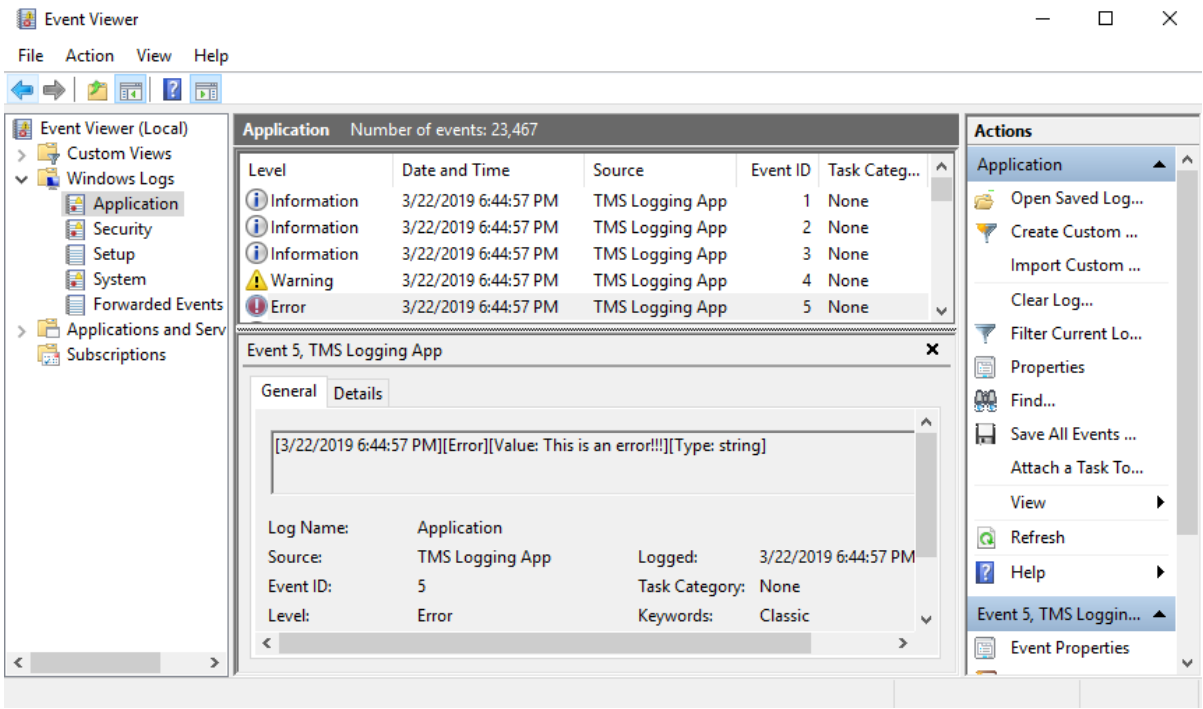
```
uses {...}, TMSLogging;

  RegisterTMSLogger;
```

Once you do the above, any log message you write will be logged using the TMSDefaultLogger global logger of TMS Logging framework, regardless of the logger name. You simply need to configure the output handlers in TMS Logging. In the following example, we configure add a Windows event output handler to TMS Logging:

```
uses
  Bcl.Logging,
  Bcl.TMSLogging,
  TMSLoggingCore,
  TMSLoggingEventLogOutputHandler;

var
  Logger: ILogger;
begin
  RegisterTMSLogger;

  // Output Log to Windows Event Log
  TMSDefaultLogger.RegisterOutputHandlerClass(
    TTMSLoggerEventLogOutputHandler, ['TMS Logging App']);

  Logger := LogManager.GetLogger;
  Logger.Error('This is an error!!!');
  Logger.Warning('Pay attention to this');
  Logger.Info('Relevant information');
  Logger.Debug('Some debug output');
  Logger.Trace('Tracing information');
end;
```

If you run the sample code above, you will see the messages logged in the Windows Event Log. Just press Windows+R, type "eventvwr" and press Enter. Or alternatively, press search for "Event Viewer" desktop application in Windows search.

Once Event Viewer is open, choose "Windows Logs" and select "Application". You should see your log messages there:



Using multiple loggers

The mode above use the single global logger. In more complex scenarios when you use multiple loggers to write log messages, you can call RegisterTMSLogger passing a anonymous function.

```
uses {...}, TMSLogging, VCL.TMSLoggingMemoOutputHandler;

  RegisterTMSLogger(procedure(const Name: string; Logger: TTMSCustomLogger)
    begin
      if Pos('Foo', Name) > 0 then
        Logger.RegisterOutputHandlerClass(
          TTMSLoggerMemoOutputHandler, [Memo1]);
      if Pos('Bar', Name) > 0 then
        Logger.RegisterOutputHandlerClass(
          TTMSLoggerMemoOutputHandler, [Memo2]);
    end
  );
```

The code above will change logging behavior and create a different TTMSLogger instance for each different named logger you use when calling GetLogger. This is useful to route different output handlers to different loggers. In the example above, if the logger name contains "Foo", any log messages generated by it will be redirected to Memo1. If the name contains Bar, log messages will be redirected to Memo2. For all other loggers, messages will be ignored.

This is very useful for you to ignore and organize log messages from different loggers, for example, you can ignore log messages coming from TMS Business loggers.

TMS Logging allows you to output the log to Text file, CSV file, TCP/IP listener, Browser, Windows Event Log, among others. For full documentation, including how to configure and use all the other available output handlers, please refer to TMS Logging manual.

About

This documentation is for TMS Sparkle.

In this section:

What's New

Version 3.14 (Aug-2020)

- **New: TStaticModule.FolderUrlMode allows you to control how redirects work in URL representing folders.**
  When user access an URL like *<server>/foldername* or *<server>/foldername/*, static module will check this property to decide if both addresses are different or one show redirect to the other.

Version 3.13 (Jun-2020)

- **Improved:** TSparkleHttpSysDispatcher.HttpSys property allows access to the internal underlying THttpSysServer object.
- **Fixed:** JWT middleware was returning status code 400 instead of 401 when the JWT sent by the client was invalid (regression).

Version 3.12 (May-2020)

- **New: Support for Delphi 10.4 Sydney.**

Version 3.11 (Apr-2020)

- **Fixed:** Sporadic Access Violation when removing a middleware from the middleware list at design time.

Version 3.10 (Mar-2020)

- **New: Encryption middleware allows using custom cryptographic algorithm to encrypt request and response data. A new demo EncryptionMiddleware is included to show the capabilities.**
- **Improved:** Multilogger allows adding more output handlers calling RegisterTMSLogger multiple times to add several output handlers at different points of application initialization.
- **Improved:** THttpResponse now implements IHttpResponse. You can use it as an interface to benefit from automatic referent counting.
- **Fixed:** THttpResponse.ContentAsBytes can now be read multiple times.
- **Fixed:** THttpRequest.RemoteIp property was returning empty when using Indy-based servers.

Version 3.9 (Nov-2019)

- **New: Support for Android 64-bit platform (Delphi 10.3.3 Rio).**
- **Fixed:** Logging middleware not logging headers (req[header-name]) in mobile platforms.

Version 3.8 (Oct-2019)

- **Fixed:** Logging middleware was not logging messages if an active TXDataServer was stopped and restarted a second time.
- **Fixed:** Logging middleware was not logging messages when the associated Sparkle module was not explicitly closing the request.

Version 3.7 (Sep-2019)

- **Improved:** Indy-based dispatcher now provides the correct URL scheme (http/https) and in consequence the correct default port (80 or 443) in the request URL. When receiving request through the Indy dispatcher, previous versions could not tell if the request was being made through HTTP or HTTPS protocol. The side effect was that the request URL schema was always being considered as HTTP and the port being 80. This is now improved, and if the request is coming via HTTPS, the default port of the request URL is now 443.

Version 3.6 (Jun-2019)

- **New: macOS 64 support in Delphi Rio 10.3.2 (version 3.6.1, Jul-2019).**
- **New: Logging mechanism now support named loggers and allow different outputs to be configured for different loggers.**
- **New: LogLevel and ExceptionLogLevel properties in TLoggingMiddleware** allows configuring the default log level for log messages generated by the middleware.
- **New: OnFilterLogEx event in logging middleware** provides additional information and allows logging messages using different log levels depending on the request/response data.
- **New: Multilogger demo shows the mechanism of using multiple named loggers.**
- **Fixed:** Workaround memory leaks in Android HTTP client for Delphi Tokyo and up.

Version 3.5 (May-2019)

- **Fixed:** WinHTTP API import function WinHttpGetProxyForUrl had an incorrect signature.

Version 3.4 (Mar-2019)

- **New: Logging framework - including TMS Logging - allows easy server-side logging to several output handlers like text files, Windows Event Log, TCP/IP listeners, etc.**
- **New: Logging middleware allows logging HTTP requests and responses processed by Sparkle servers.** It uses the new logging framework and can automatically log requests/responses in details with a few lines of code.

Version 3.3 (Jan-2019)

- **New: Design-time Generic Middleware allows adding middleware custom code** using the design-time middleware editor, as a RAD alternative to creating a custom middleware from code.
- **New: TSparkleHttpSysDispatcher.OnStart and OnStop events.**
- **Improved:** Design-time middleware editor now displays the name of middleware components in the list, avoiding ambiguous names.

Version 3.2 (Dec-2018)

- **New: THttpClient.OnResponseReceived event, fires when an HTTP response is received from server.**

Version 3.1 (Nov-2018)

- **New: Support for Delphi 10.3 Rio.**
- **New: Sample project for file upload handling.** A new project FileUploadSample included in demos folder shows how to use TMultipartFormDataReader to receive files from clients.
- **Improved:** TStaticModule.LastModified property enables the inclusion of Last-Modified header in the response (improves caching).
- **Fixed:** TMultipartFormDataPart.FileName property automatically dequotes string.

Version 3.0 (Oct-2018)

- **New: Design-time support with new set of components for Sparkle: TSparkleGenericServer and TSparkleStaticServer components, and design-time middleware editor.**
- **New: Smooth support for handling multipart/form-data content with new TMultipartFormDataReader class.**
- **Fixed:** TIndySparkleHTTPServer not properly handling requests with content-type application/x-www-form-urlencoded.
- **Fixed:** JWT expiration dates were not being saved in UTC.

Version 2.7 (Sep-2018)

- **New: TJwtMiddleware.AllowExpiredToken and ForbidAnonymousAccess properties.** This makes it easier to reject requests with expired tokens or requests without tokens (anonymous) just by setting those properties. This is a breaking change as the middleware now rejects expired tokens by default.
- **New: TCorsMiddleware middleware** makes it straightforward to add CORS support to any Sparkle module.
- **Improved:** THttpSysContext.SysRequest property made public allowing low-level access to http.sys raw request
- **Improved:** Wizard now adds (commented) CORS and Compress middleware in source code. Uncomment to use them.
- **Improved:** Better error messages for iOS/Mac http client

Version 2.6 (May-2018)

- **New:** THttpSysServer.KeepHostInUrlPrefixes property.
- **Improved:** Windows HTTP client now decompresses deflate/gzip responses automatically at OS level (Windows 8.1 and up).
- **Improved:** THttpClient automatically decompresses responses encoded with gzip or deflate.
- **Improved:** Documentation update with example on how to ignore self-signed certificates when using HTTP client on Windows.
- **Fixed:** THttpResponse.ContentLength returning wrong value on Windows when response body size was greater than 2GB.

Version 2.5 (Feb-2018)

- **New: TIndySparkleHTTPServer Indy-based web server component allows use of XData/RemoteDB and any other Sparkle-based module to use Indy.**
  This is an alternative for http.sys-based or Apache-based Sparkle server, useful when you want to have a stand-alone Sparkle/XData/RemoteDB server on platforms like Linux (without Apache) or even Android.
  Thanks to Michael Van Canneyt for his contribution on this feature!
- **Fixed:** HEAD requests sent by the server were causing a timeout in the very next request sent by the same client.
- **Fixed:** Memory leak in Compress middleware if client was disconnected before server response was completed.
- **Fixed:** Range Check Error in PercentEncode function when compiler range check option was enabled.

- **Fixed:** Error not found (404) when request URL had a query parameter with an URL address (for example, *"https://mysparkleserver.com/oauth/login?redirect_uri=https://service.google.com"*).

Version 2.4.2 (Oct-2017)

- **Fixed:** ContentLength was being set to 0 when calling THttpServerResponse.Close with no content. This would cause HEAD responses to not return correct content-length in some situations. Now ContentLength will only be set to 0 if it was not previously modified.

Version 2.4.1 (Jul-2017)

- **Fixed:** Sparkle/XData servers retrieving wrong content from client request on Linux using Web Broker (regression).

Version 2.4 (Jul-2017)

- **Improved:** THttpServerRequest.ContentStream now provides correct value for Size property if request has a ContentLength header.
- **Fixed:** When port of requested URL was different from port of server side module URL, the module was responding with 404 (not found) response.
- **Fixed:** Sparkle HTTP Client was is now merging multiple HTTP headers (with same name) sent by the server. Previously it was keeping only the last one.

Version 2.3 (May-2017)

- **New: Linux support using Delphi 10.2 Tokyo and later.** You can now use Sparkle modules on Linux, through Apache modules created using WebBroker.
- **New: In-process server.** You can now create an in-process server that will handle client requests from same application without needing to create a real HTTP server using http.sys or Apache. This is very useful for building stand-alone applications using Sparkle servers without changing your existing code, and it's also very useful to easily test your server modules.
- **Fixed:** Error when using some UTF8 characters in URL like "ç" or "ã", even when percent encoded.
- **Fixed:** HTTP client issue on Android in Delphi Tokyo.
- **Fixed:** Memory leak in Android HTTP client.

Previous Versions

Version 2.2 (Mar-2017)

- New: Delphi 10.2 Tokyo Support.

- New: Option for custom proxy configuration for Windows clients or automatic proxy detection on Windows 8.1 and later.

Version 2.1 (Aug-2016)

- New: TStaticModule for serving static files.

Version 2.0.1 (Jul-2016)

- Improved: Minor source code modifications for compatibility with TMS XData 2.1.

Version 2.0 (May-2016)

- New: Authentication and authorization mechanism.
- New: JWT (JSON Web Token) authentication middleware (XE6 and up).
- New: Basic authentication middleware.
- New: JSON Web Token library for creating and validating tokens (based on the Delphi JOSE and JWT Library: http://github.com/paolo-rossi/delphi-jose-jwt).
- New: Delphi 10.1 Berlin support.
- New: IsFloat and IsInt64 methods in TJsonReader class.
- Improved: Json Reader and Writer supports reading/writing literal values (JSON values that are neither object or array).

Version 1.4 (Feb-2016)

- New: Design-time wizard to create a Sparkle Server easily with a few clicks.
- New: Middleware system provides a modular way to add custom request and response processing.
- New: Compress Middleware provides automatic gzip/deflate response based on request headers.
- New: TWinHttpRequest.BeforeWinHttpSendRequest event allow low level setting of WinHttp options.
- Fixed: Error when sending non-chunked responses with size greater than 2GB.
- Fixed: Response.Close not working correctly with chunked responses.

Version 1.3.1 (Sep-2015)

- New: Delphi 10 Seattle support.

Version 1.3 (Aug-2015)

- New: THttpClient.OnSendingRequest event provides an opportunity to customize request, for example adding custom headers.

Version 1.2.4 (Jun-2015)

- Improved: Better handling of URI percent encoding/decoding.

Version 1.2.3 (Apr-2015)

- New: Delphi XE8 support.

Version 1.2.2 (Mar-2015)

- Improved: Support THttpMethod.Options new type allows easier implementation of handling Http OPTIONS requests.
- Fixed: THttpServerRequest.RawMethod property empty when request had non-standard http methods.
- Fixed: Support for modules that respond to root URL addresses.

Version 1.2.1 (Dec-2014)

- Fixed: Issues with floating-point numbers in JSON reader/writer in non-English systems.
- Fixed: Mac/iOS HTTP Client was setting content-type to x-www-form-urlencoded automatically (inconsistent with other clients).

Version 1.2 (Oct-2014)

- New: TMSHttpConfig tool to configure URL reservations and HTTPS certificates using a friendly GUI. Source code included as a demo.
- New: THttpSysServerConfig class methods and properties for configuring URL reservation and HTTPS certificates from Delphi source code.
- New: TAnonymousServerModule class to easily create a server modules with very simple request processing.
- Improved: TJsonReader doesn't raise an exception if input stream is empty, it stays in EOF state instead.
- Fixed: Possible wrong module routing when modules share same path segments at different ports.

Version 1.1.1 (Sep-2014)

- New: Delphi XE7 support.

Version 1.1 (Aug-2014)

- New: JSON classes for reading and writing JSON now fully documented.
- New: THttpServerRequest.RemoteIP provides the server with the IP address of client which sent request to the server.
- New: THttpRequest.Timeout property allows specifying a different timeout value for each request.
- New: Patch value in THttpMethod enumeration type to check for PATCH requests.
- Improved: Error messages now explain detailed reason why HTTPS connection fails in Windows clients either due like invalid server certificate, expired certificate, or other reasons.
- Improved: More detailed error messages in JSON reader when reading fails.
- Fixed: JSON writer wrongly escaping of Unicode control characters.

Version 1.0.1 (May-2014)

- New: Delphi XE6 support.

Version 1.0 (Mar-2014)

- First public release.

Licensing and Copyright Notice

The trial version of this product is intended for testing and evaluation purposes only. The trial version shall not be used in any software that is not run for testing or evaluation, such as software running in production environments or commercial software.

For use in commercial applications or applications in production environment, you must purchase a single license, a small team license or a site license. A site license allows an unlimited number of developers within the company holding the license to use the components for commercial application development and to obtain free updates and priority email support for the support period (usually 2 years from the license purchase). A single developer license allows ONE named developer within a company to use the components for commercial application development, to obtain free updates and priority email support. A small team license allows TWO developers within a company to use the components for commercial application development, to obtain free updates and priority email support. Single developer and small team licenses are NOT transferable to another developer within the company or to a developer from another company. All licenses allow royalty free use of the components when used in binary compiled applications.

The component cannot be distributed in any other way except through TMS Software web site. Any other way of distribution must have written authorization of the author.

Online registration/purchase for this product is available at https://www.tmssoftware.com. Source code & license is sent immediately upon receipt of payment notification, by email.

Getting Support

General notes

Before contacting support:

- Make sure to read this whole manual and any readme.txt or install.txt files in component distributions, if available.
- Search TMS support forum and TMS newsgroups to see if you question hasn't been already answer.
- Make sure you have the latest version of the component(s).

When contacting support:

- Specify with which component is causing the problem.
- Specify which Delphi or C++Builder version you're using and preferably also on which OS.
- For registered users, use the special priority support email address (mentioned in registration email) & provide your registration email & code. This will guarantee the fastest route to a solution.

Send email from an email account that

1. allows to receive replies sent from our server;
2. allows to receive ZIP file attachments;
3. has a properly specified & working reply address.

Getting support

For general information: info@tmssoftware.com

Fax: +32-56-359696

For all questions, comments, problems and feature request for our products: help@tmssoftware.com

---

IMPORTANT

All topics covered by this manual are officially supported and it's unlikely that future versions will break backward compatibility. If this ever happens, all breaking changes will be covered in this manual and guidelines to update to a new version will be described. However, it's important to note that parts of the source code of this product that are undocumented are not officially supported and are **subject to change**, which includes breaking backward compatibility. In case you are using an unsupported/undocumented feature we will not provide support for upgrading and will not officially support it.

Breaking Changes

List of changes in each version that breaks backward compatibility from a previous version.

Version 2.7

JWT Middleware now rejects expired tokens by default, for security reasons. You can optionally turn off this behavior when creating the JWT Middleware by passing True to the AAllowExpiredTokens parameter of Create constructor.

Online Resources

This topic lists some links to internet resources - videos, articles, blog posts - about TMS Sparkle.

Official Online Documentation:
http://www.tmssoftware.biz/business/sparkle/doc/web/

Blog Posts:

Hello, TMS Sparkle
https://www.tmssoftware.com/site/blog.asp?post=284