Overview

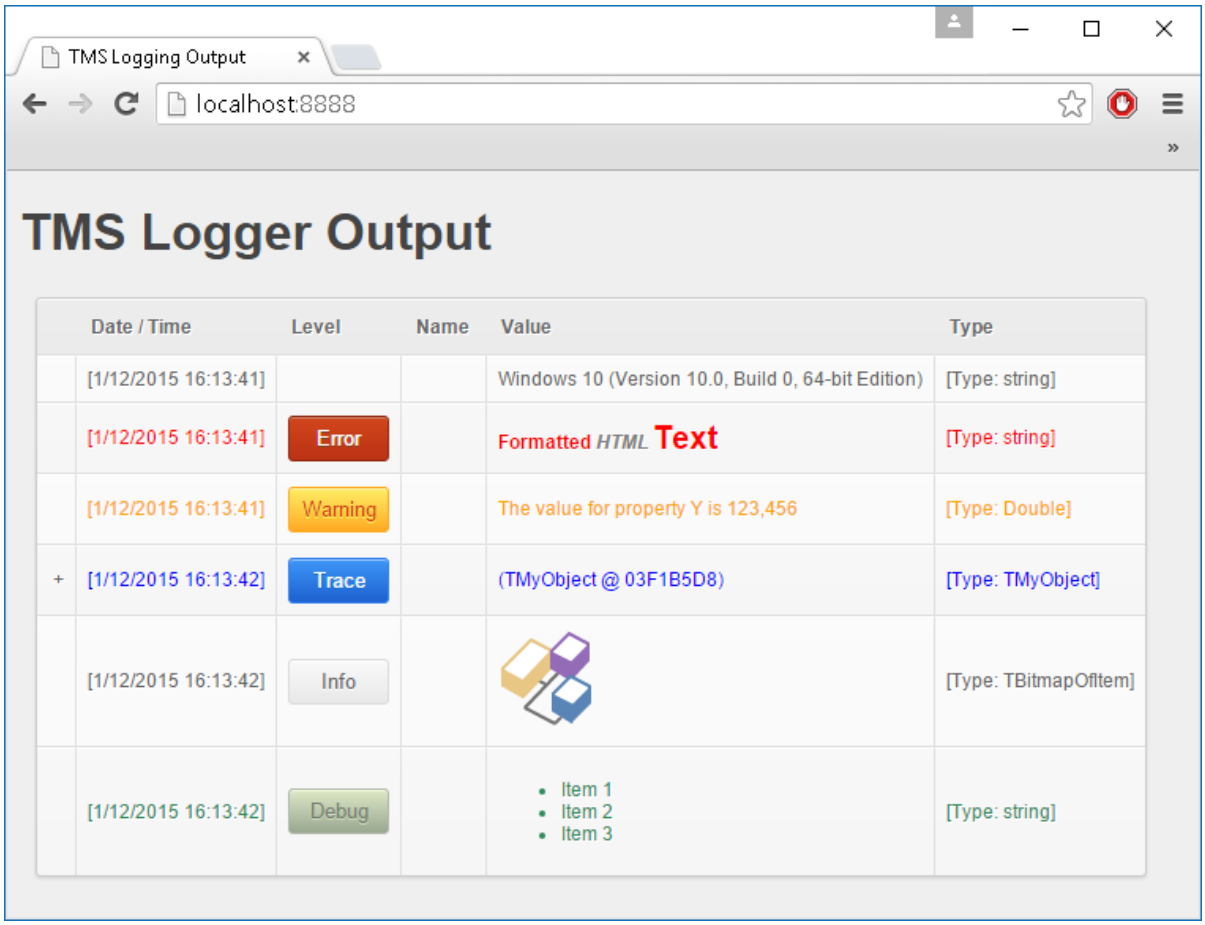
TMS Logging is a cross platform logging framework offering informative log output to a flexible number of targets with a minimum amount of code. It's part of TMS Business product line. TMS Logging is available for XE7 update 1 or newer releases and supports VCL and FMX.

TMS Logging product page: https://www.tmssoftware.com/site/tmslogging.asp

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

Feature overview

- Log to one or more output handlers such as the Console, HTML, Text file, CSV file, TCP/IP, Browser, Windows Event Log, ...
- Heavily RTTI based for comprehensive type and class logging with simple log statements
- Cross platform: supports VCL Win32/Win64 apps and FMX Win32/Win64/Mac OS-X/iOS/Android apps
- Class & property attribute based log output control & log output validation
- Extensive & extensible data formatting capabilities
- Multi-thread enabled & thread-safe
- Includes options for time & delta time measurements
- Runtime configurable log level
- Log configuration persistence to file or registry
- Helper methods to quickly setup custom output handlers and retrieve important information on the machine, device and application
- Value validations to control logging based on attributes with a set of pre-defined validations such as value-range, date/time range, string length, regular expressions, ...
- Easily extensible and customizable with custom output handlers
- Separate TCP/IP Client included for viewing logger outputs remotely
- IDE Plugin for adding missing units, inserting output handler registration code and toggling comments



In this section:

Using TMSLogger

TMSLogger: getting started, features and capabilities.

Output Handlers

How to output log to different targets and formats.

Miscelaneous

Some additional features.

Using TMSLogger

Add the unit `VCL.TMSLogging` or `FMX.TMSLogging` to the uses list depending on the kind of framework you are creating an application for. Additionally, the units `TMSLoggingCore` and `TMSLoggingUtils` are only neccessary, for example when setting the Outputs or Filters property. The TMSLoggingCore and TMSLoggingUtils are shared between VCL and FMX. The IDE Plugin that is available after installation can help you add the missing units in case the application does not compile after adding code related to TMS Logging. More information can be found in the IDE Plugin chapter.

The function TMSLogger returns a singleton logger instance that can be used throughout the application. By default the logger is active, but can easily be deactivated by using the following code:

```
TMSLogger.Active := False;
```

As already mentioned in the overview topic, the logger makes use of log levels to output values / objects. Each call is a combination of the log level, an optional format and the values / objects to log. The logger outputs to the console by default, and already applies a set of outputs (TMSLogger.Outputs) with a specific format (TMSLogger.OutputFormats). Below is a sample of different log levels with the default logger configuration.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'Hello World !';
  TMSLogger.Info(s);
  TMSLogger.Error(s);
  TMSLogger.Warning(s);
  TMSLogger.Trace(s);
  TMSLogger.Debug(s);
end;
```

Result:

```
Debug Output: [12/1/2015 12:14:03 PM][Info][Value: Hello World !][Type: string] Process Demo.exe (3768)
Debug Output: [12/1/2015 12:14:03 PM][Error][Value: Hello World !][Type: string] Process Demo.exe (3768)
Debug Output: [12/1/2015 12:14:03 PM][Warning][Value: Hello World !][Type: string] Process Demo.exe (3768)
Debug Output: [12/1/2015 12:14:03 PM][Trace][Value: Hello World !][Type: string] Process Demo.exe (3768)
Debug Output: [12/1/2015 12:14:03 PM][Debug][Value: Hello World !][Type: string] Process Demo.exe (3768)
```

The TMSLogger provides several other features and capabilities you can use, as follows.

Formatting

Formatting can be applied in 2 ways: either globally with the TMSLogger.OutputFormats properties or by using one of the log level overloads. The parsing after applying formatting is a custom implementation and supports the Delphi SysUtils Format function. A requirement to successfully execute a log statement with parsing is that the format string contains an opening and closing brace or curly bracket to form a format tag.

The TMSLogger.OutputFormats property contains a set of predefined formats with their format tags. Below is an overview of the default values for each output format:

```
TimeStampFormat := '[{%dt}]';
ProcessIDFormat := '[{%s}]';
ThreadIDFormat := '[{%s}]';
LogLevelFormat := '[{%s}]';
ValueFormat := '[Value: {%s}]';
NameFormat := '[Name: {%s}]';
TypeFormat := '[Type: {%s}]';
MemoryUsageFormat := '[Memory usage: {%bt} bytes]';
```

Accompanied with these format properties is the TMSLogger.Outputs property that can determine which information needs to be logged. Below is a sample that combines these 2 properties to create a completely different log output.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TMSLogger.Outputs := [loTimeStamp, loLogLevel, loValue];
  TMSLogger.OutputFormats.TimeStampFormat := 'The time is {%"hh:nn:ss"dt}, ';
  TMSLogger.OutputFormats.LogLevelFormat := 'the loglevel is {%s}, ';
  TMSLogger.OutputFormats.ValueFormat := 'and the value is {%s}';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'Hello World !';
  TMSLogger.Info(s);
  TMSLogger.Error(s);
  TMSLogger.Warning(s);
  TMSLogger.Trace(s);
  TMSLogger.Debug(s);
end;
```

Result will be as following:

```
Debug Output: The time is 12:14:48, the loglevel is Info, and the value is Hello World ! Process Demo.exe (4116)
Debug Output: The time is 12:14:48, the loglevel is Error, and the value is Hello World ! Process Demo.exe (4116)
Debug Output: The time is 12:14:48, the loglevel is Warning, and the value is Hello World ! Process Demo.exe (4116)
Debug Output: The time is 12:14:48, the loglevel is Trace, and the value is Hello World ! Process Demo.exe (4116)
Debug Output: The time is 12:14:48, the loglevel is Debug, and the value is Hello World ! Process Demo.exe (4116)
```

Overriding format for a specific log message

While the above method of formatting already provides a certain flexibility, the value formatting is applied to each log statement. To override this behavior, you can specify a formatting for each value that will be logged. The following sample overrides the output of the previous sample for one of the log statements.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TMSLogger.Outputs := [loTimeStamp, loLogLevel, loValue];
  TMSLogger.OutputFormats.TimeStampFormat := 'The time is {%"hh:nn:ss"dt}, ';
  TMSLogger.OutputFormats.LogLevelFormat := 'the loglevel is {%s}, ';
  TMSLogger.OutputFormats.ValueFormat := '{%s}';
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
  fmt: string;
begin
  s := 'Hello World !';
  fmt := 'The value is {%s}';
  TMSLogger.Info(s);
  TMSLogger.Error(s);
  TMSLogger.WarningFormat(fmt, [s]);
  TMSLogger.Trace(s);
  TMSLogger.Debug(s);
end;
```

Result will be as following:

```
Debug Output: The time is 12:15:24, the loglevel is Info, and the value is Hello World ! Process Demo.exe (4416)
Debug Output: The time is 12:15:24, the loglevel is Error, and the value is Hello World ! Process Demo.exe (4416)
Debug Output: The time is 12:15:24, the loglevel is Warning, and the value is The value is Hello World ! Process Demo.exe (4416)
Debug Output: The time is 12:15:24, the loglevel is Trace, and the value is Hello World ! Process Demo.exe (4416)
Debug Output: The time is 12:15:24, the loglevel is Debug, and the value is Hello World ! Process Demo.exe (4416)
```

Format string syntax

Formatting is not limited to strings only, below is an overview of supported formatting tags that can be used.

Tag	Expected value
d	Decimal (integer)
e	Scientific
f	Fixed
g	General
m	Money

Tag	Expected value
n	Number (floating)
p	Pointer
s	String
u	Unsigned decimal
x	Hexadecimal

The general format of each formatting tag is as follows:

```
{%[Index:][-][Width][.Precision]Tag}
```

where the square brackets refer to optional parameters, and the **:.-** characters are literals, the first 2 of which are used to identify two of the optional arguments. Below is an example of formatting a double value with 2 decimals:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  d: Double;
  fmt: string;
begin
  d := 123.456;
  fmt := 'The value is {%.2f}';
  TMSLogger.InfoFormat(fmt, [d]);
end;
```

Result will be as following:

```
Debug Output: [12/1/2015 12:16:12 PM][Info][Value: The value is 123.46][Type: Double] Process Demo.exe (3504)
```

More information can be found at the following page: http://www.delphibasics.co.uk/RTL.asp?Name=Format

As an extension to this, the logger supports a set of additional tags that can be used to format the value. Below is an overview of the format tags that can be used.

Tag	Optional parameters (between double quotes)	Expected value	Output
a		array	outputs the array to a string that represents the values
b	output as number or as "true" or "false" string (ex: {%"-1"b})	Boolean	outputs the Boolean value to a string represented in numbers or "true" or "false" string

Tag	Optional parameters (between double quotes)	Expected value	Output
bin	number of decimals (ex: {%"8"bin}	TStream object	outputs the TStream object as a binary formatted string
bt	output as data size (B, KB, MB, GB, TB) and optional decimals separated with '#' and formatted with the FormatFloat function (ex: '{%"GB#0.00000"bt}'	Ordinal	outputs the value as a data size formatted value
dt	datetime format (ex: {%"mm-dd-yyyy"dt)	TDateTime	outputs the TDateTime value to a string
hex	number of digits (ex: {%"2"hex}	TStream object	outputs the TStream object as a hex formatted string
pic		object with image data	outputs image data as a string that is sent to the output handlers
pichex		object with image data	outputs image data as a hex string that is sent to the output handlers
st		TStream object	outputs the TStream object as a string

A sample using one of the above tags is demonstrated in the following sample:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  fmt: string;
begin
  fmt := 'Today is {%"ddd, dd mmm yyyy"dt}';
  TMSLogger.InfoFormat(fmt, [Now]);
end;
```

Output:

```
Debug Output: [12/1/2015 12:16:38 PM][Info][Value: Today is Tue, 01 Dec 2015][Type: Double] Process Demo.exe (5232)
```

OnCustomFormat event

When the above format tags are not sufficient to output the value, the logger exposes an OnCustomFormat event that passes the format tag it has received and the value it needs to parse. The var AResult parameter of this event can be returned based on the custom format tag. Below is an example which demonstrates this.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  fmt: string;
  d: Double;
begin
  d := 123.456;
  fmt := 'This is a custom tag with value {%CT}';
  TMSLogger.InfoFormat(fmt, [d]);
end;

procedure TForm1.DoCustomFormat(Sender: TObject; AValue: TValue;
  AFormat: string; var AResult: string);
begin
  if AFormat.ToUpper.Contains('CT') and AValue.IsType<Double> then
    AResult := FloatToStr(AValue.AsType<Double>);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  TMSLogger.OnCustomFormat := DoCustomFormat;
end;
```

Output:

```
Debug Output: [12/1/2015 12:17:39 PM][Info][Value: This is a custom tag with value 123.456][Type: Double] Process Demo.exe (6116)
```

The above sample simply verifies whether the custom tag is used. The AFormat parameter contains the tag part of the format string that we pass as a parameter to our TMSLogger.InfoFormat call. The AFormat parameter value is {%CT}. The AValue parameter contains the Double value and the AResult var parameter is assigned a simple FloatToStr of the AValue parameter.

Validations

Validations are attributes (based on the Delphi attribute concept) in which a certain comparison is made and a Boolean is returned whether that condition is met. When the result from this comparison is true, the log output will be sent to the output handlers. By default, there are no validations applied, thus the output is always logged.

When monitoring a certain value, be it a string, a Double, an integer, etc. value, and you only want to output this value when it matches, for example, a string length, a regular expression, or when the value exceeds a minimum or maximum, you can create a validation attribute and add it as a parameter of one of the logger calls. The `TMSLoggingCore` unit already provides a set of validation classes that are ready to use. Below is an overview of those validation classes and a short explanation.

- *TMSLoggerRangeValidation*: returns true if the value that needs to be logged exceeds a certain minimum and maximum.

- *TMSLoggerDateTimeValidation*: returns true if the value that needs to be logged exceeds a certain start and end date. The date parameters when creating a TMSLoggerDateTimeValidation attribute are strings and are converted with the DateTimeToStr function.
- *TMSLoggerStringValidation*: returns true if the value doesn't match or contain a certain string.
- *TMSLoggerStringLengthValidation*: returns true if the value doesn't match or exceed a certain string length.
- *TMSLoggerRegularExpressionValidation*: returns true if the value doesn't match a certain regular expression.

When the condition is met, the value is logged. Each validation has a set of parameters to further fine-tune the condition. Two important properties are the ReverseCondition and the Format properties. When the ReverseCondition property is true, the condition is reversed, for example in case of the TMSLoggerRangeValidation, the condition returns true if the value is within a certain minimum and maximum. The format property can be used to apply a certain formatting when the condition is met and the value is logged. Below is a sample that demonstrates the use of a validation attribute and the ReverseCondition and Format properties.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  fmt: string;
  i: Integer;
  vl: TMSLoggerRangeValidation;
begin
  fmt := 'The value is {%g}';
  vl := TMSLoggerRangeValidation.Create(110, 130);
  vl.Format := fmt;
  i := 100;
  TMSLogger.Info(i, [], [vl]);
  i := 122;
  TMSLogger.Info(i, [], [vl]);
  i := 140;
  TMSLogger.Info(i, [], [vl]);
  i := 110;
  TMSLogger.Info(i, [], [vl]);
  vl.Free;
end;
```

Output:

```
Debug Output: [12/1/2015 12:18:22 PM][Info][Value: The value is 100][Type: Integer] Process Demo.exe (4540)
Debug Output: [12/1/2015 12:18:22 PM][Info][Value: The value is 140][Type: Integer] Process Demo.exe (4540)
```

Note that with the above sample, only the values 100 and 140 will be logged, because they exceed the minimum and maximum values that were set as parameters of the TMSLoggerRangeValidation attribute. Setting the ReverseCondition to True, will generate a different output as demonstrated in the following sample.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  fmt: string;
  i: Integer;
  vl: TMSLoggerRangeValidation;
begin
  fmt := 'The value is {%g}';
  vl := TMSLoggerRangeValidation.Create(110, 130);
  vl.Format := fmt;
  vl.ReverseCondition := True;
  i := 100;
  TMSLogger.Info(i, [], [vl]);
  i := 122;
  TMSLogger.Info(i, [], [vl]);
  i := 140;
  TMSLogger.Info(i, [], [vl]);
  i := 110;
  TMSLogger.Info(i, [], [vl]);
  vl.Free;
end;
```

Output:

```
Debug Output: [12/1/2015 12:18:47 PM][Info][Value: The value is 122][Type: Integer] Process Demo.exe (4276)
Debug Output: [12/1/2015 12:18:47 PM][Info][Value: The value is 110][Type: Integer] Process Demo.exe (4276)
```

In this case, the values 122 and 110 will be logged, because they are within the minimum and maximum values and the reverse condition flag is set to True. Note that the Format parameter is set to 'The value is {%g}' which will then output the values with this specific format. The format string can also be directly passed as a parameter to the InfoFormat call as already demonstrated in one of the previous samples.

Custom validations

When the default validation attributes are not sufficient, you can create your own validation by inheriting from the TMSLoggerBaseValidation attribute class and overriding the function Validate(AValue: TValue): Boolean. Below is a sample that demonstrates this.

```
type
  MyValidation = class(TMSLoggerBaseValidation)
  protected
    function Validate(AValue: TValue): Boolean; override;
  end;

implementation

procedure TForm1.Button1Click(Sender: TObject);
var
  d: Double;
  vl: MyValidation;
begin
  vl := MyValidation.Create;
  d := 3;
  TMSLogger.Debug(d, [], [vl]);
  d := 4.5;
  TMSLogger.Debug(d, [], [vl]);
  vl.Free;
end;

{ MyValidation }

function MyValidation.Validate(AValue: TValue): Boolean;
begin
  Result := False;
  if AValue.IsType<Double> then
    Result := Frac(AValue.AsType<Double>) = 0;
end;
```

Output:

```
Debug Output: [12/1/2015 12:19:23 PM][Debug][Value: 4.5][Type: Double] Process Demo.exe (5144)
```

The sample returns a true if the fractional part of the value is 0 which means that the value 3 is a valid value and will not be logged. The properties to control formatting and reverse conditions are not available by default when inheriting from TMSLoggerBaseValidation. When these properties and functionality need to be available in your custom validation class, you can inherit from TMSLoggerValidation instead.

Logger output statements are not limited to only one validation. Multiple validation attributes can be passed as a parameter. The logger calls have an array of TMSLoggerBaseValidation parameter that can contain multiple values. Only when each validation condition is met, the value will be logged.

Using declarative attributes

As explained earlier, the validation attributes are based on the Delphi attribute concept which means that they can also be added to an object's properties. This way, the logger can simply pass the complete object as a parameter, and the values will be logged when the validation condition is met. Below is a sample that demonstrates this.

```
type
  TMyObject = class
  private
    FX: string;
    FY: Double;
  public
    [TMSLoggerStringLengthValidation(5)]
    property X: string read FX write FX;
    [TMSLoggerRangeValidation(10, 20)]
    property Y: Double read FY write FY;
  end;

implementation

procedure TForm1.Button1Click(Sender: TObject);
var
  obj: TMyObject;
begin
  obj := TMyObject.Create;
  obj.X := 'Hello';
  obj.Y := 30;
  TMSLogger.Warning(obj);
  obj.Free;
end;
```

Output:

```
Debug Output: [12/1/2015 12:20:03 PM][Warning][Value: (TMyObject @ 035CACC0)][Type: TMyObject] Process Demo.exe (408)
Debug Output:   [12/1/2015 12:20:03 PM][Warning][Name: Y][Value: 30][Type: Double] Process Demo.exe (408)
```

The output of this sample exists of the object itself and the Y property. The X property is not logged, because the 'Hello' string has a length of 5.

The use of validation attributes require the `TMSLoggingCore` unit to be added to the uses list. If the warning below occurs after compilation, then the attribute is not found and will not be detected by the logger.

```
[dcc32 Warning] UDemo.pas(14): W1025 Unsupported language feature: 'custom
attribute'
```

Property Filtering

The logger supports filtering based on the visibility of the field or property that is being logged. The Filters property can be used to determine if public and/or published properties need to be logged with or without attributes. This way, the logger can analyze and only log the field or property that match the filter. The filter is set to allow all public and published properties with attributes by default. Below is a sample that demonstrates the use of the Filter property.

```
type
  TMyObject = class
  private
    FZ: string;
    FX: Double;
    FY: Integer;
  public
    property X: Double read FX write FX;
    [TMSLoggerRangeValidation(0, 10)]
    property Y: Integer read FY write FY;
  published
    property Z: string read FZ write FZ;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var
  obj: TMyObject;
begin
  TMSLogger.Filters := [lfPublic, lfPublished];
  obj := TMyObject.Create;
  obj.X := 3.456;
  obj.Y := 10;
  obj.Z := 'Hello World';
  TMSLogger.Warning(obj);
  obj.Free;
end;
```

Output:

```
Debug Output: [12/1/2015 12:21:15 PM][Warning][Value: (TMyObject @ 097B7C90)][Type: TMyObject] Process Demo.exe (2284)
Debug Output:   [12/1/2015 12:21:15 PM][Warning][Name: X][Value: 3.456][Type: Double] Process Demo.exe (2284)
Debug Output:   [12/1/2015 12:21:15 PM][Warning][Name: Z][Value: Hello World][Type: string] Process Demo.exe (2284)
```

Notice that the output only shows the X and Z properties, because the Filters property is set the only allow public and published properties without attributes. The Y property has an attribute and therefore not logged. The attributed validates a range between 0 and 10 and the value of the Y property is valid, and thus not logged. If the value would exceed the range, the value would be logged but only if the filters are modified to allow public properties with attributes.

Using attributes

Additionally, filtering can be fine-tuned with attributes. The filter attributes add the same kind of filtering as on logger level. The logger parses the attributes and determines if the sub properties / fields of the class or property that has the attribute applied, are valid for logging. There are 2 kinds of filter attributes, *TMSLoggerClassFilter* and *TMSLoggerPropertyFilter*. Below is a sample that demonstrates this.

```
type
  [TMSLoggerClassFilter([lfPublic, lfPublicWithAttributes])]
  TMyObject = class
  private
    FZ: string;
    FX: Double;
    FY: Integer;
  public
    property X: Double read FX write FX;
    [TMSLoggerRangeValidation(0, 10)]
    property Y: Integer read FY write FY;
  published
    property Z: string read FZ write FZ;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var
  obj: TMyObject;
begin
  obj := TMyObject.Create;
  obj.X := 3.456;
  obj.Y := 12;
  obj.Z := 'Hello World';
  TMSLogger.Warning(obj);
  obj.Free;
end;
```

Output:

```
Debug Output: [12/1/2015 12:21:54 PM][Warning][Value: (TMyObject @ 035F4660)][Type: TMyObject] Process Demo.exe (4160)
Debug Output:   [12/1/2015 12:21:54 PM][Warning][Name: X][Value: 3.456][Type: Double] Process Demo.exe (4160)
Debug Output:   [12/1/2015 12:21:54 PM][Warning][Name: Y][Value: 12][Type: Integer] Process Demo.exe (4160)
```

In this case, only the X and Y properties will be logged, because the *TMSLoggerClassFilter* attribute specifies to allow public properties with and without attributes. The value Y is 12 in this case, exceeds the range validation and is logged. If the value would be set to 10, as in the previous sample, the Y property would also not be logged.

The use of filter attributes requires the `TMSLoggingCore` unit to be added to the uses list. If the warning below occurs after compilation, then the attribute is not found and will not be detected by the logger.

```
[dcc32 Warning] UDemo.pas(14): W1025 Unsupported language feature: 'custom
attribute'
```

Multi-value Logging

The logger has a set of overloads per log level that can be used to format the output, specify validation attributes for conditional logging and specify an array of property names when an object is logged. One of the overloads is designed to quickly log a set of objects in a single call. Below is a sample that demonstrates this.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a: string;
  b: Double;
  c: Boolean;
begin
  a := 'Hello World';
  b := 4.56;
  c := True;
  TMSLogger.DebugValues([a, b, c]);
end;
```

Output:

```
Debug Output: [12/1/2015 12:22:40 PM][Debug][Value: Hello World][Type: string] Process Demo.exe (4948)
Debug Output: [12/1/2015 12:22:40 PM][Debug][Value: 4.56][Type: Double] Process Demo.exe (4948)
Debug Output: [12/1/2015 12:22:40 PM][Debug][Value: True][Type: Boolean] Process Demo.exe (4948)
```

Note that this call does not have a separate format parameter. The formatting is based on the OutputFormats property.

Timing

By default the logger outputs the current date/time, but has the capability of formatting the timestamp output as milliseconds, microseconds or ticks depending on the *TimeStampOutputMode* property. When using an output mode other than the default value for this property, you always need to combine the logger calls with a StartTimer / StopTimer.

Microseconds

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  TMSLogger.TimeStampOutputMode := tsomMicroseconds;
  TMSLogger.StartTimer;
  Sleep(5);
  TMSLogger.Debug(1);
  Sleep(15);
  TMSLogger.Debug(2);
  Sleep(5);
  TMSLogger.Debug(3);
  Sleep(5);
  TMSLogger.Debug(4);
  TMSLogger.StopTimer;
end;
```

Output:

```
Debug Output: [4280 µs][Debug][Value: 1][Type: Integer] Process Demo.exe (5736)
Debug Output: [19221 µs][Debug][Value: 2][Type: Integer] Process Demo.exe (5736)
Debug Output: [23679 µs][Debug][Value: 3][Type: Integer] Process Demo.exe (5736)
Debug Output: [27917 µs][Debug][Value: 4][Type: Integer] Process Demo.exe (5736)
```

Delta Microseconds

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  TMSLogger.TimeStampOutputMode := tsomMicrosecondsDelta;
  TMSLogger.StartTimer;
  Sleep(5);
  TMSLogger.Debug(1);
  Sleep(15);
  TMSLogger.Debug(2);
  Sleep(5);
  TMSLogger.Debug(3);
  Sleep(5);
  TMSLogger.Debug(4);
  TMSLogger.StopTimer;
end;
```

Output:

```
Debug Output: [4989 µs][Debug][Value: 1][Type: Integer] Process Demo.exe (4844)
Debug Output: [14875 µs][Debug][Value: 2][Type: Integer] Process Demo.exe (4844)
Debug Output: [4372 µs][Debug][Value: 3][Type: Integer] Process Demo.exe (4844)
Debug Output: [4191 µs][Debug][Value: 4][Type: Integer] Process Demo.exe (4844)
```

Direct timing

Additionally, timing can be done with the StartTimer and StopTimer independent of the TimeStampOutputMode as demonstrated in the following sample:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  TMSLogger.StartTimer;
  Sleep(500);
  TMSLogger.StopTimer(True, lsmMilliseconds, 'The elapsed time is {%d} ms');
end;
```

Output:

```
Debug Output: [12/1/2015 12:24:57 PM][Value: The elapsed time is 500 ms][Type: Int64] Process Demo.exe (4304)
```

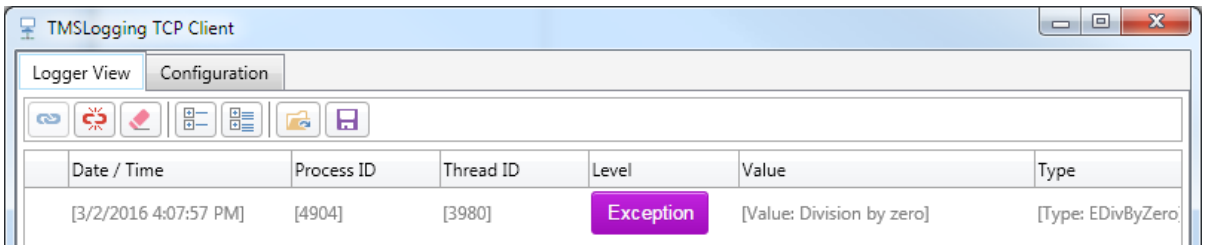
Exceptions

The logger supports automatic handling and logging of exceptions. As an addition to the standard log levels, an exception log level is added to the set. By default, exception handling is not enabled. To enable it set *ExceptionHandling* to true on logger level.

```
TMSLogger1.ExceptionHandling := True;
```

Whenever an exception occurs, it will be automatically logged with an Exception log level, as demonstrated in the following division by zero exception.

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerTCPOutputHandler, [Self]);
TMSLogger.ExceptionHandling := True;
TMSLogger.Outputs := AllOutputs;

procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c: Integer;
begin
  a := 10;
  b := 0;
  c := a div b;
end;
```



Enabling exception handling will create a TApplicationEvents object in VCL and assign the Application.OnException event. If your application needs to handle additional code when an exception occurs the logger exposes an *OnHandleException* event.

HTML Support

The Browser Output and HTML Output handlers are both using HTML to display the output information. To add additional formatting inside the table or plain HTML formatted viewer, HTML tags can be added to the log statements. Below is a sample that demonstrates this.

```
procedure TForm1.Button3Click(Sender: TObject);
var
  d: Double;
begin
  d := 4.5;
  TMSLogger.DebugFormat('The <span style=''color:red''><b>value</b></span> is
{%f}', [d]);
end;
```

Even when specifying HTML tags, the other non-HTML formatted output handlers will still display the content correct, as they strip HTML when receiving output information from the logger.

Other TTMSLogger methods

The TMSLogger object has a set of helper methods that can be used to output additional information to the output handlers. Below is an overview of each method and a short explanation.

Name	Description
Clear	Sends a clear instruction to each output handler and removes the log files / log data.
GetTimer(const AMode: TTMSLoggerTimerMode = lsmMilliseconds; const ALogResult: Boolean = False; const AFormat: string = ''): Int64	Returns an elapsed value in ticks or milliseconds based on the mode parameters after a timer has been started with StartTimer.
Indent	Increases the indent for log statements.
IsTimerRunning	Returns a Boolean if the timer is still running after it was started with the StartTimer call.
LogCurrentDateTime(const AFormat: string = '')	Logs the current date / time with an optional formatting parameter.
LogCurrentLocale(const AFormat: string = '');	Logs the current language identifier.
LogProcessID(const AFormat: string = '');	Logs the process id.
LogScreenShot	Logs a screenshot of the main form of the application.

Name	Description
LogScreenShot(const AControl: TControl)	Logs a screenshot of the specified control.
LogSeparator	Logs a separator string.
LogSystemInformation(const AFormat: string = '')	Logs the system information of the operating system.
LogThreadID(const AFormat: string = '');	Logs the thread id.
LogMemoryUsage(const AFormat: string = '');	Logs the memory usage of the application.
LogMemoryUsageDifference(const ALogResult: Boolean = True; const AFormat: string = ''): Cardinal	Logs the difference of the memory usage of the application based on the previous call to LogMemoryUsage.
StartTimer	Starts a timer, needs to be paired with StopTimer.
StopTimer(const AMode: TTMSLoggerTimerMode = lsmMilliseconds; const ALogResult: Boolean = False; const AFormat: string = ''): Int64	Returns an elapsed value in ticks or milliseconds based on the mode parameters and stops the timer, needs to be paired with StartTimer.
Unindent	Decreases the indent for log statements.

Output Handlers

The logger provides the capability of logging to various other output handlers. An output handler is a class that provides a log output method with a parameter that contains output information. The output information contains the timestamp, the value, the log level and a set of pre-formatted strings (such as the name, type and value, based on the TMSLogger.Outputs property) and then saves the information to a file, or sends it to a browser, TCP/IP client. The output handlers support various formats such as plain text and HTML.

Each output handler except for the default TTMSLoggerConsoleOutputHandler class is available in separate units. Each unit name starts with *TMSLogging* and then specifies which output handler is implemented. To use an output handler, it needs to be registered first. The logger has a set of *RegisterOutputHandler** methods that can be used to register or create an instance of an output handler.

An output handler has an Active property, which is true by default and has a set of constructor overloads. The *RegisterOutputHandlerClass* method will accept a list of parameters that should match the number of parameters in the constructor of the output handler class you wish to create. When you are not sure on the type of parameters, you can take a look at the create signature, or simply create a separate instance of the output handler. When creating a separate instance, it can be registered using the *RegisterOutputHandler* or *RegisterManagedOutputHandler* call instead. Using the latter will tell the logger to destroy the output handler instance when the logger instance itself is destroyed. The former will not destroy the output handler.

Each output handler has a procedure LogOutput(const AOutputInformation: TTMSLoggerOutputInformation); that contains the log information.

Following are the output handlers that are currently available.

Console Output

Class: *TTMSLoggerConsoleOutputHandler* at `TMSLoggingCore` unit.

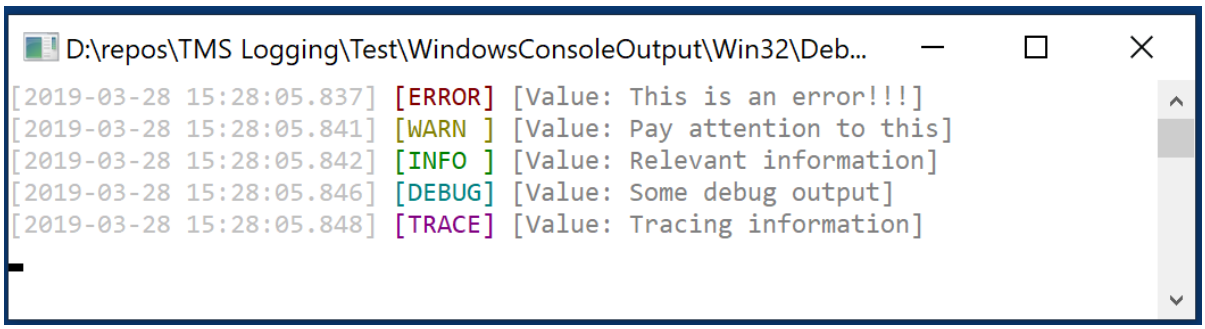
The *TTMSLoggerConsoleOutputHandler* outputs the log information to the console window of the IDE or the log monitor/console of the device (iOS, Android, Mac OSX). Note that this does not output to the regular Windows console. To do that, you should use the Windows Console Output handler.

```
Debug Output: [11/12/2015 11:38:10 AM][Value: Windows 7 Service Pack 1 (Version 6.1, Build 7601, 64-bit Edition)][Type: string] Process Demo.exe (5548)
Debug Output: [11/12/2015 11:38:10 AM][Error][Value: Formatted HTML Text][Type: string] Process Demo.exe (5548)
Debug Output: [11/12/2015 11:38:10 AM][Warning][Value: The value for property Y is 123.456][Type: Double] Process Demo.exe (5548)
Debug Output: [11/12/2015 11:38:10 AM][Trace][Value: (TMyObject @ 039CB218)][Type: TMyObject] Process Demo.exe (5548)
Debug Output:   [11/12/2015 11:38:10 AM][Trace][Name: X][Value: Hello World !][Type: string] Process Demo.exe (5548)
Debug Output:   [11/12/2015 11:38:10 AM][Trace][Name: Y][Value: 123.456][Type: Double] Process Demo.exe (5548)
Debug Output: [11/12/2015 11:38:10 AM][Value: Elapsed time: 19 ms][Type: Integer] Process Demo.exe (5548)
```

Windows Console Output

Class name: *TTMSLoggerWindowsConsoleOutputHandler* at `TMSLoggingWindowsConsoleOutputHandler` unit.

The *TTMSLoggerWindowsConsoleOutputHandler* outputs log messages to the Microsoft Windows console window. It also changes the output color according to the log level.

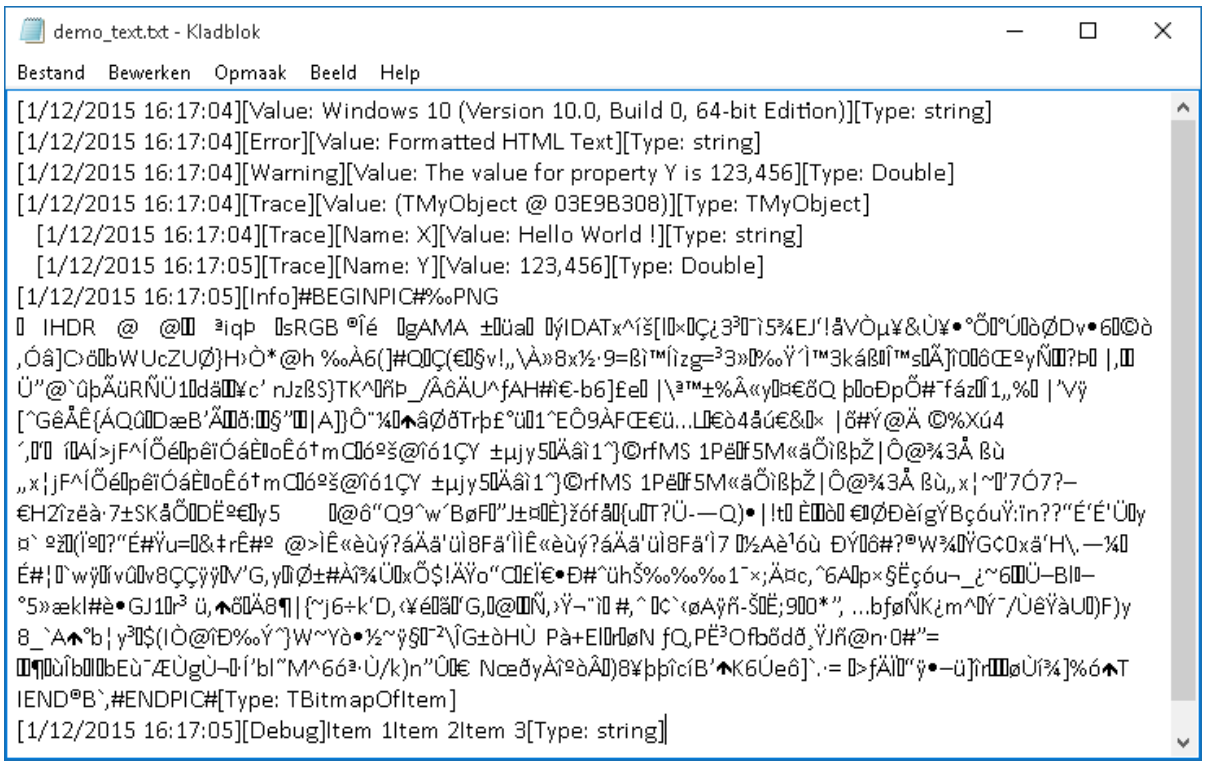


Text File Output

Class name: *TTMSLoggerTextOutputHandler* at `TMSLoggingTextOutputHandler` unit.

The *TTMSLoggerTextOutputHandler* outputs the information as plain text to a file.

Sample output:



Event Log Output

Class name: *TTMSLoggerEventLogOutputHandler* at `TMSLoggingEventLogOutputHandler` unit.

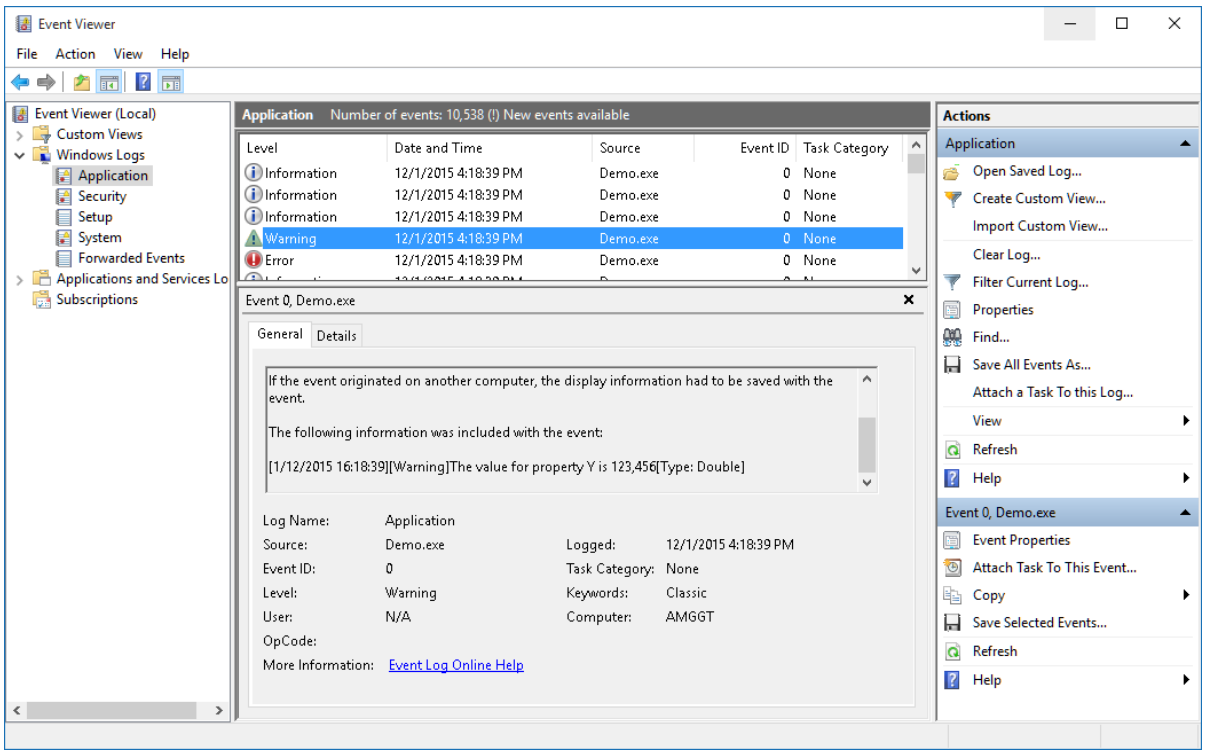
The *TTMSLoggerEventLogOutputHandler* will log the output information to the Windows event log. Creating an instance can be done with the following code:

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerEventLogOutputHandler);
```

IMPORTANT

The TTMSLoggerEventOutputHandler requires administrator privileges on Windows in order to successfully write an event to the Windows event log. Starting the application without administrator privileges will also log to the event log but will display a message indicating it cannot find the event id linked to the specific event log.

Sample output:

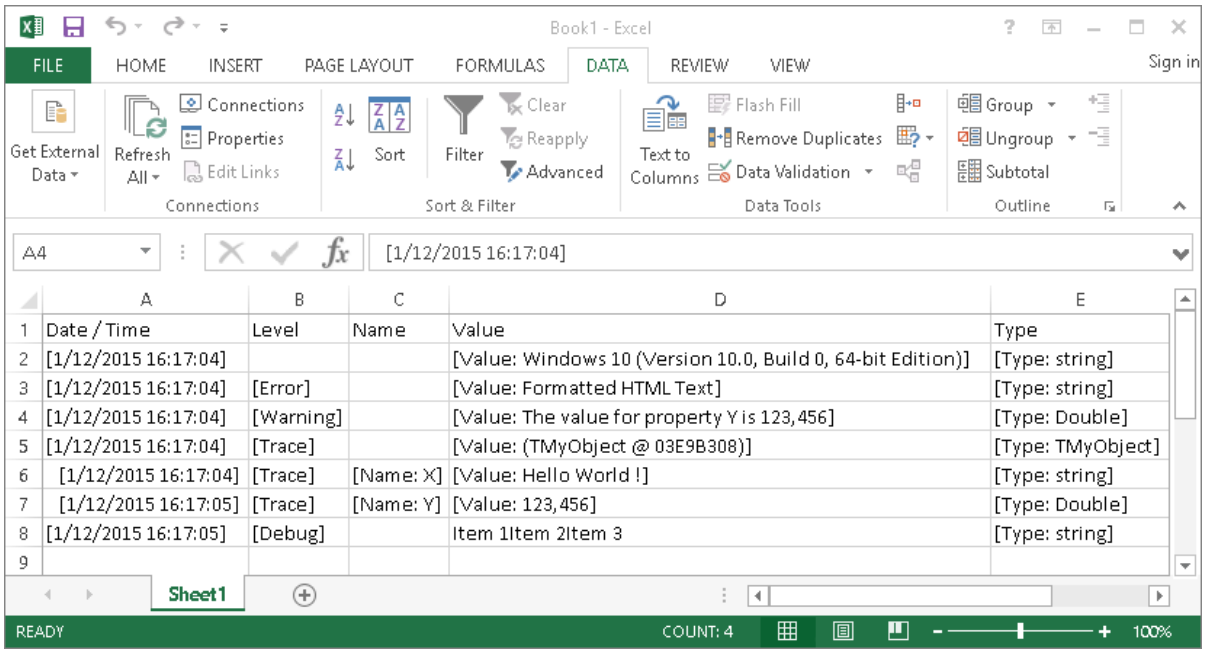


CSV File Output

Class name: *TTMSLoggerCSVOutputHandler* at `TMSLoggingCSVOutputHandler` unit.

The *TTMSLoggerCSVOutputHandler* outputs the information as plain text to a CSV file, separated by a delimiter, which can be configured with a separate property.

Sample output:



	A	B	C	D	E
1	Date / Time	Level	Name	Value	Type
2	[1/12/2015 16:17:04]			[Value: Windows 10 (Version 10.0, Build 0, 64-bit Edition)]	[Type: string]
3	[1/12/2015 16:17:04]	[Error]		[Value: Formatted HTML Text]	[Type: string]
4	[1/12/2015 16:17:04]	[Warning]		[Value: The value for property Y is 123,456]	[Type: Double]
5	[1/12/2015 16:17:04]	[Trace]		[Value: (TMyObject @ 03E9B308)]	[Type: TMyObject]
6	[1/12/2015 16:17:04]	[Trace]	[Name: X]	[Value: Hello World !]	[Type: string]
7	[1/12/2015 16:17:05]	[Trace]	[Name: Y]	[Value: 123,456]	[Type: Double]
8	[1/12/2015 16:17:05]	[Debug]		Item 1Item 2Item 3	[Type: string]
9					

TCP Output

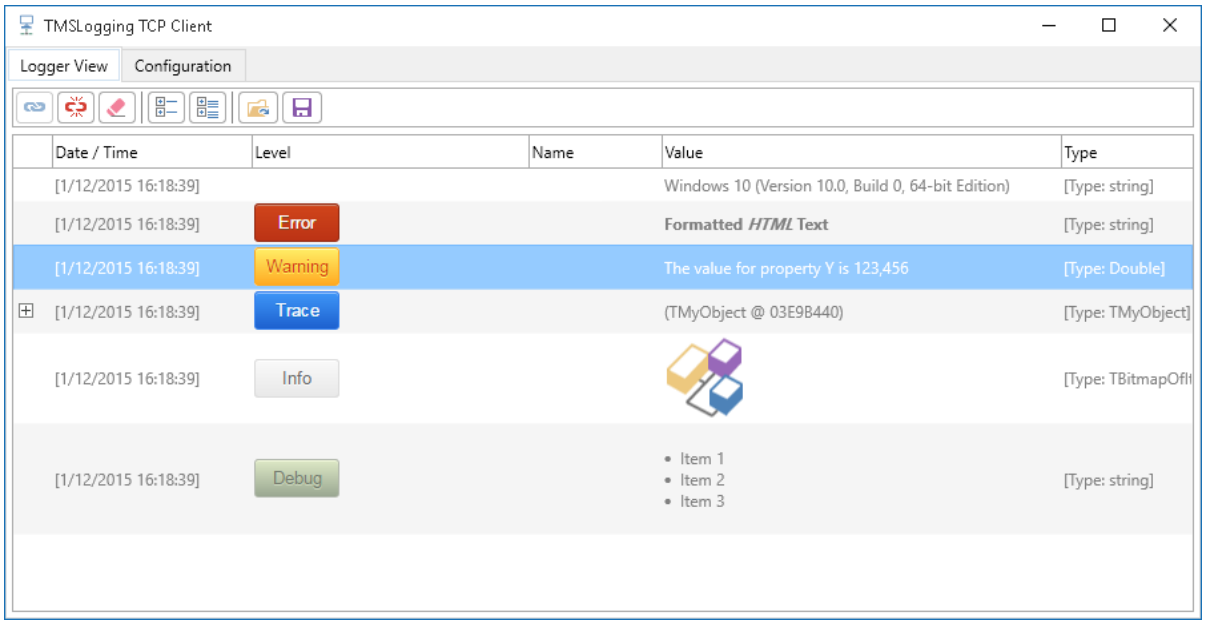
Class name: *TTMSLoggerTCPOutputHandler* at `TMSLoggingTCPOutputHandler` unit.

The *TTMSLoggerTCPOutputHandler* is a server that sends the output information to each connected TCP/IP client. The client application can implement its own TCP/IP client read buffer instructions but an instance of TTMSLoggerTCPClient, which already implements this, can also be chosen. The TTMSLoggerTCPClient has an OnReceivedOutputInformation event is triggered whenever output information is received.

Registering a TTMSLoggerTCPOutputHandler is similar to the TTMSLoggerBrowserOutputHandler, but does not have an option to change the HTML formatting viewer.

The sample output screenshot is a TCP Client that implements this technique and serves as a viewer. The executable is available in the installation directory and is supported for Windows and Mac OSX.

Sample output:



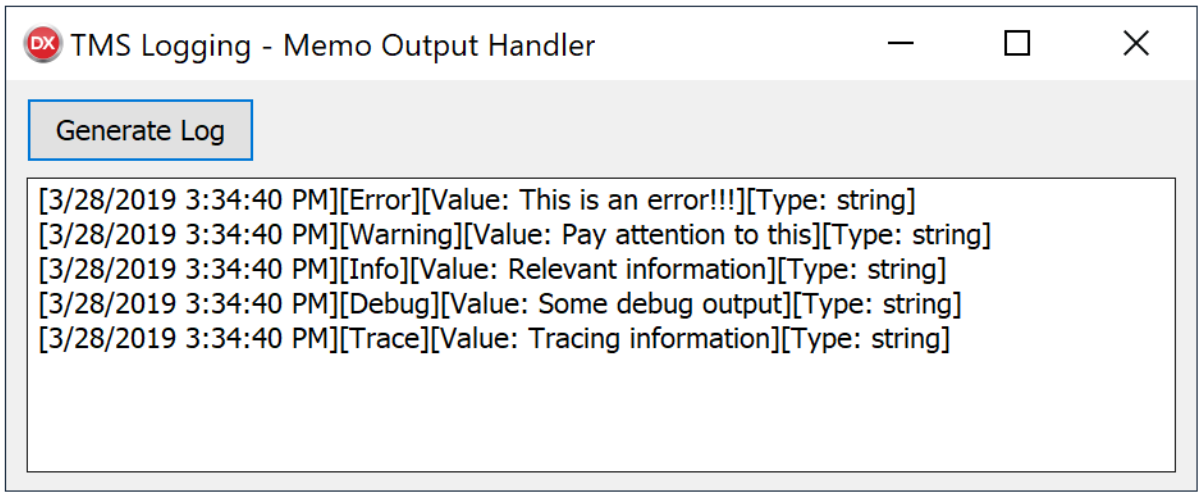
Memo Output

Class name: *TTMSLoggerMemoOutputHandler* at `Vcl.TMSLoggingMemoOutputHandler` unit.

The *TTMSLoggerMemoOutputHandler* outputs log messages to a TMemo VCL control, in a thread-safe way. It's useful to allow the end-user to quickly visualize log information in the VCL application.

The Create constructor requires you to pass the instance of TMemo that will receive the messages.

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerMemoOutputHandler, [Memo1]);
```



Browser Output

Class name: *TTMSLoggerBrowserOutputHandler* at `TMSLoggingBrowserOutputHandler` unit.

The *TTMSLoggerBrowserOutputHandler* is a server that sends the output information to each connected client browser. To register a TTMSLoggerBrowserOutputHandler, the following code can be used:

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerBrowserOutputHandler, [Self]);
```

After registering it, navigating to the IP-address of the server, or the localhost with the default port of 8888 will display an empty HTML table. As soon as the logger logs a value, the table will be updated. To change the port number, the parameter list can be modified as demonstrated in the code below.

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerBrowserOutputHandler, [Self, 1234]
);
```

Optionally, the HTML table view can be changed to an HTML plain view by specifying an additional parameter:

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerBrowserOutputHandler, [Self, 1234,
TValue.From(ohmPlain)]);
```

Available constructors:

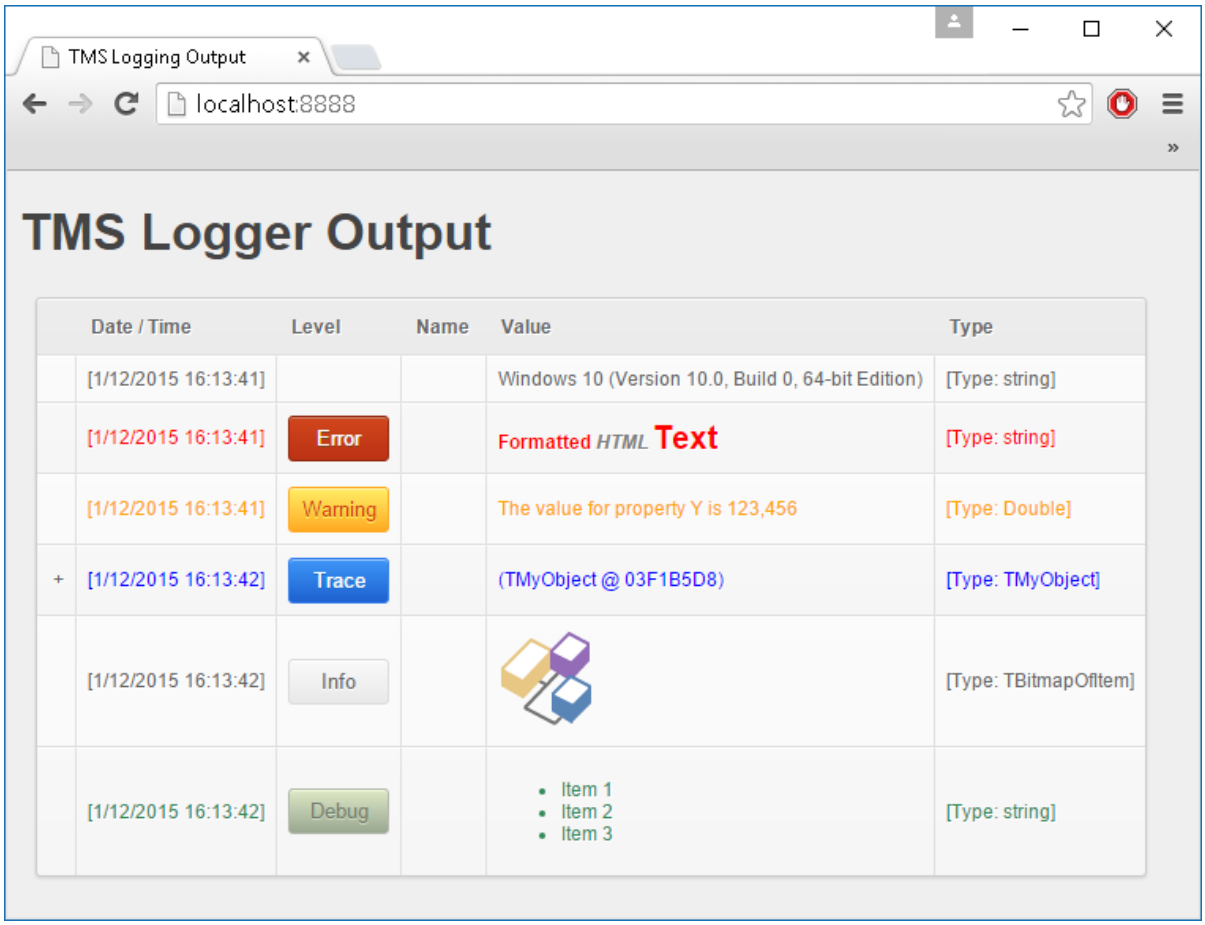
```
constructor Create(const AOwner: TComponent); reintroduce; overload; virtual;

constructor Create(const AOwner: TComponent; const APort: Integer); reintroduce;
overload; virtual;

constructor Create(const AOwner: TComponent; const APort: Integer; const AMode: T
TMSLoggerHTMLOutputHandlerMode); reintroduce; overload; virtual;

constructor Create(const AOwner: TComponent; const APort: Integer; const AMode: T
TMSLoggerHTMLOutputHandlerMode; const ATitle: string); reintroduce; overload; vir
tual;
```

- *AOwner*: A component to serve as the owner for internal components. Usually Self.
- *APort*: The TCP port used to create the TCP/IP listener
- *AMode*: The format data will be output.
- *ATitle*: The HTML page title.

Example:

```
var
  Handler: TTMSLoggerBrowserOutputHandler;
begin
  Handler := TTMSLoggerBrowserOutputHandler.Create(Self);
  TMSLogger.RegisterManagedOutputHandler(Handler);
  Handler.Title := 'Browser Logging (Custom Title)';
```

Sample output:



HTML Output

Class name: *TTMSLoggerHTMLOutputHandler* at `TMSLoggingHTMLOutputHandler` unit.

The *TTMSLoggerHTMLOutputHandler* is based on the same output as the TTMSLoggerBrowserOutputHandler but saves the output information to a file instead. The constructor overloads for this class are different than the ones for the TTMSLoggerBrowserOutputHandler. To know exactly which parameters to pass to the TMSLogger.RegisterOutputHandlerClass, you can simply type "TTMSLoggerHTMLOutputHandler.Create(" which will show a list of constructor overloads.

Available constructors:

```
constructor Create(const AFileName: string); overload; override;

constructor Create(const AFileName: string; ADataName: string); reintroduce; over
load; virtual;

constructor Create(const AFileName: string; ADataName: string; AMode: TTMSLoggerH
TMLOutputHandlerMode); reintroduce; overload; virtual;

constructor Create(const AFileName: string; ADataName: string; AMode: TTMSLoggerH
TMLOutputHandlerMode; const ATitle: string); reintroduce; overload; virtual;
```

- *AFileName*: The name of .html file to be generated.

- *ADataName*: The name of .js file to be generated.
- *AMode*: The format data will be output.
- *ATitle*: The HTML page title.

Examples:

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerHTMLOutputHandler, ['.
\demo_table.html', 'demo_table.js', TValue.From(ohmTable), 'Table logging']);
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerHTMLOutputHandler, ['.
\demo_plain.html', 'demo_plain.js', TValue.From(ohmPlain), 'Plain logging']);
```

Datasource Output

Class name: *TTMSLoggerDataSourceOutputHandler* at `TMSLoggingDataSourceOutputHandler` unit.

The *TTMSLoggerDataSourceOutputHandler* is capable of connecting to a datasource via the register method demonstrated in the code below.

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerDataSourceOutputHandler, [DataSour
ce1]);
```

The result of the registration returns an instance of TTMSLoggerDataSourceOutputHandler which contains a Fields property to setup the field names to connect to via the datasource parameter. By default these fields are already prefilled with property name as a field name.

Aurelius Output

Class name: *TTMSLoggerAureliusOutputHandler* at `TMSLoggingAureliusOutputHandler` unit.

The *TTMSLoggerAureliusOutputHandler* connects via an Aurelius IDBConnectionPool interface to send logging outputs to a dataset connected via TMS Aurelius. The process of setting up a connection is simple as demonstrated in the code below.

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerAureliusOutputHandler, [TValue.Fro
m<IDBConnectionPool>(pool)]);
```

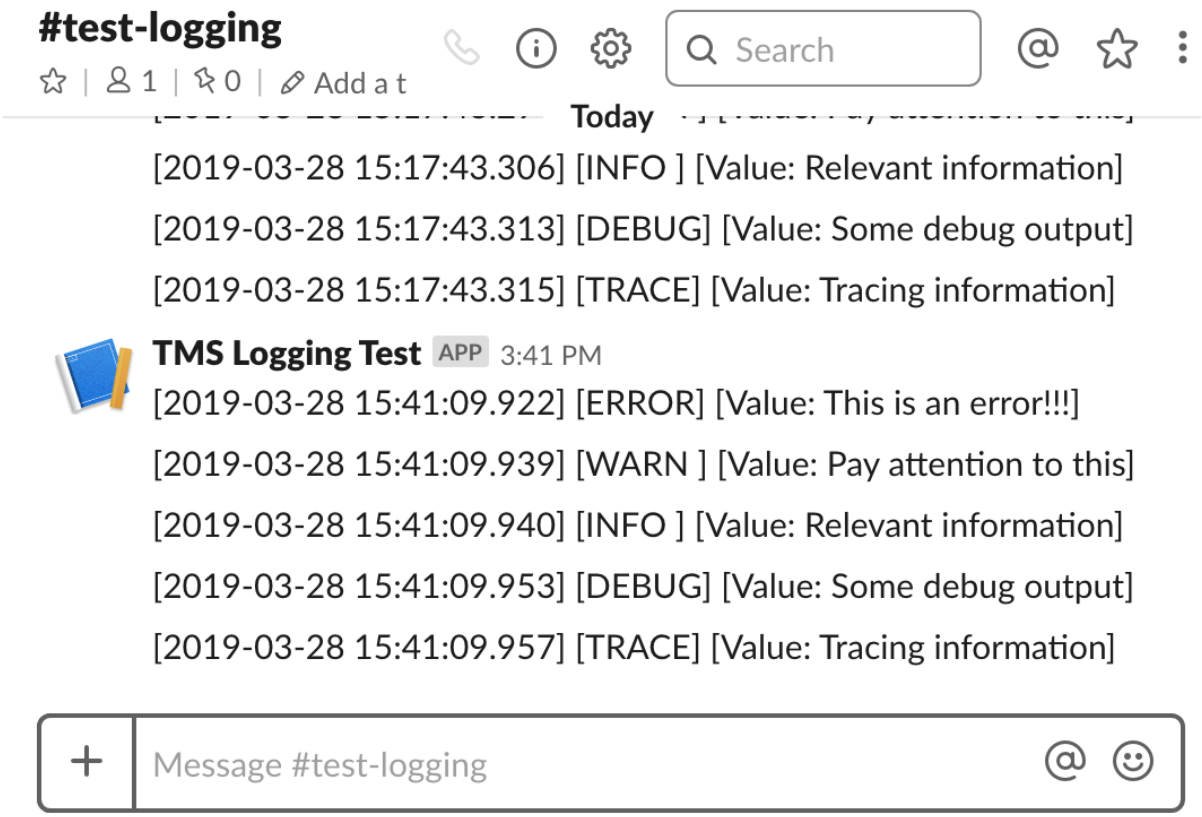
Slack Output

Class name: *TTMSLoggerSlackWebhookOutputHandler* at `TMSLoggingSlackOutputHandler` unit.

The *TTMSLoggerSlackWebhookOutputHandler* outputs log messages to a Slack channel through an incoming webhook. The incoming webhook is a Slack feature which provides you a unique URL that is used to post messages to a specific channel in your workplace.

The Create constructor requires you to pass the incoming webhook URL:

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerSlackWebhookOutputHandler, ['https
://hooks.slack.com/services/ABCDEFG/ABCDEFG/hajsdfhkadsfjbXBAHASQW']);
```



Exceptionless Output

Class name: *TTMSLoggerExceptionlessOutputHandler* at `VCL.TMSLoggingExceptionlessOutputHandler` or `FMX.TMSLoggingExceptionlessOutputHandler` unit.

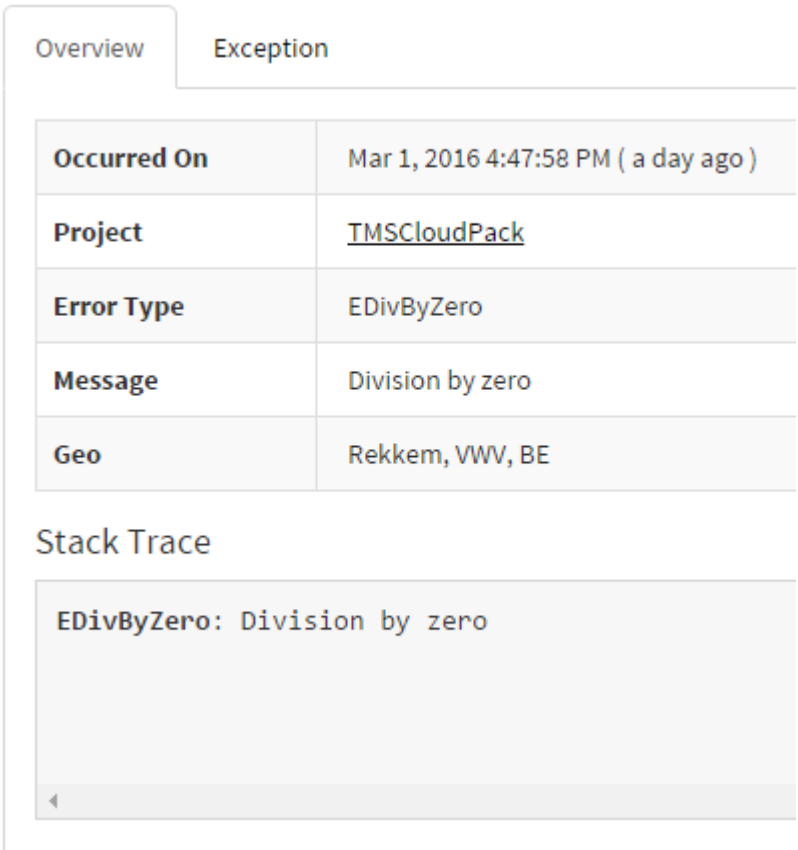
The *TTMSLoggerExceptionlessOutputHandler* connects via the TAdvExceptionless (VCL) or TTMSFMXCloudExceptionless (FMX) non-visual component. When registering this output handler, the log statements are send to the Exceptionless server which can be monitored through the dashboard service at https://exceptionless.com. This output handler is not pre-installed, and requires the TMS Cloud Pack for VCL and/or the TMS Cloud Pack for FMX installed. To start working with TTMSLoggerExceptionlessOutputHandler add the unit for the framework you are using in your application and add the following to initialize:

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerExceptionlessOutputHandler, [AdvEx
ceptionLess1, 'AProjectID']);
```

or

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerExceptionlessOutputHandler, [TMSFM
XCloudExceptionLess1, 'AProjectID']);
```

AProjectID is a string that needs to be replaced after the Exeptionless cloud component has retrieved its projects. The output handler requires an already authenticated Exceptionless cloud component in order to function properly.



myCloudData Output

Class name: *TTMSLoggerMyCloudDataOutputHandler* at `VCL.TMSLoggingMyCloudDataOutputHandler` or `FMX.TMSLoggingMyCloudDataOutputHandler` unit.

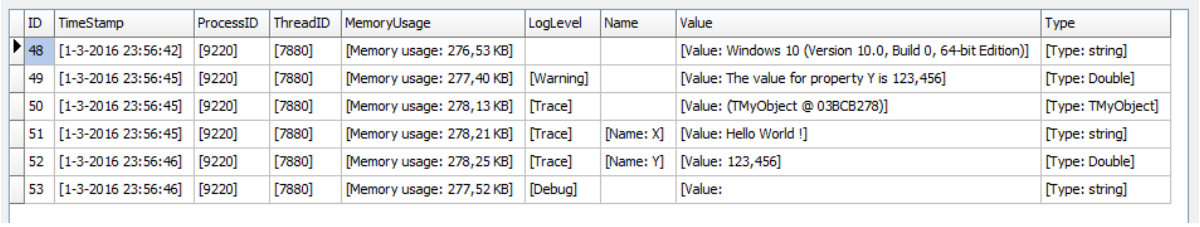
The *TTMSLoggerMyCloudDataOutputHandler* connects via the TAdvMyCloudData (VCL) or TTMSFMXCloudMyCloudData (FMX) non-visual component. When registering this outputhandler, the log statements are send to the MyCloudData server. This outputhandler is not pre-installed, and requires the TMS Cloud Pack for VCL and/or the TMS Cloud Pack for FMX installed. To start working with TTMSLoggerMyCloudDataOutputHandler add the unit for the framework you are using in your application and add the following to initialize:

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerMyCloudDataOutputHandler, [AdvMyCl
oudData1, 'ATableName']);
```

or

```
TMSLogger.RegisterOutputHandlerClass(TTMSLoggerMyCloudDataOutputHandler, [TMSFMXC
loudMyCloudData1, 'ATableName']);
```

The *ATableName* string parameter is optional. By default the output handler will try to create a table called 'MyCloudDataLogging' and add Meta data through the MetaData property. This property can be accessed after registering the TTMSLoggerMyCloudDataOutputHandler which returns an instance. The MetaData properties are initialized with a default value that can be overridden. Below is a sample screenshot after following the above steps. The default values for the table and Meta data names are used in tis sample.



ID	TimeStamp	ProcessID	ThreadID	MemoryUsage	LogLevel	Name	Value	Type
48	[1-3-2016 23:56:42]	[9220]	[7880]	[Memory usage: 276,53 KB]			[Value: Windows 10 (Version 10.0, Build 0, 64-bit Edition)]	[Type: string]
49	[1-3-2016 23:56:45]	[9220]	[7880]	[Memory usage: 277,40 KB]	[Warning]		[Value: The value for property Y is 123,456]	[Type: Double]
50	[1-3-2016 23:56:45]	[9220]	[7880]	[Memory usage: 278,13 KB]	[Trace]		[Value: (TMyObject @ 03BCB278)]	[Type: TMyObject]
51	[1-3-2016 23:56:45]	[9220]	[7880]	[Memory usage: 278,21 KB]	[Trace]	[Name: X]	[Value: Hello World !]	[Type: string]
52	[1-3-2016 23:56:46]	[9220]	[7880]	[Memory usage: 278,25 KB]	[Trace]	[Name: Y]	[Value: 123,456]	[Type: Double]
53	[1-3-2016 23:56:46]	[9220]	[7880]	[Memory usage: 277,52 KB]	[Debug]		[Value:	[Type: string]

Custom Output

If the default output handlers are not sufficient, the logger supports implementing a custom output handler. The *TTMSLoggerBaseOutputHandler* and *TTMSLoggerCustomFileOutputHandler* class provides a set of procedures to easily implement a custom output handler. The *TTMSLoggerCustomFileOutputHandler* class inherits from *TTMSLoggerBaseOutputHandler* and adds support for logging to a file. The functionality to concatenate the output information and strip the HTML is located in the `TMSLoggingUtils` unit, and can be access with *TTMSLoggerUtils* class functions and procedures. Below is a sample that outputs the log information inside a TMemo based on the *TTMSLoggerBaseOutputHandler*:

NOTE

The code below is just for learning purposes. It's simple and not thread-safe to make it easier to understand. If you want log to a TMemo control, use the built-in Memo Output handler.

```
type
  TMyOutputHandler = class(TTMSLoggerBaseOutputHandler)
  private
    FMemo: TMemo;
  protected
    procedure Clear; override;
    procedure LogOutput(const AOutputInformation: TTMSLoggerOutputInformation); o
verride;
  public
    constructor Create(const AMemo: TMemo); reintroduce; virtual;
    property Memo: TMemo read FMemo write FMemo;
  end;

implementation

{ TMyOutputHandler }

procedure TMyOutputHandler.Clear;
begin
  inherited;
  if Assigned(Memo) then
    Memo.Lines.Clear;
end;

constructor TMyOutputHandler.Create(const AMemo: TMemo);
begin
  inherited Create;
```

```
  FMemo := AMemo;
end;

procedure TMyOutputHandler.LogOutput(const AOutputInformation: TTMSLoggerOutputIn
formation);
begin
  inherited;
  if Assigned(Memo) then
    Memo.Lines.Add(TTMSLoggerUtils.StripHTML(
      TTMSLoggerUtils.GetConcatenatedLogMessage(AOutputInformation, True)));
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  TMSLogger.RegisterOutputHandlerClass(TMyOutputHandler, [Memo1]);
end;
```

Miscelaneous

TMS Logging provides some additional features and capabilities, assorted listed below.

Helper procedures

In `TMSLoggingUtils` there are some general-purpose procedures and functions that you can use:

Name	Description
AddBackslash(const AValue: string): string	Returns the string with a backslash if the string does not contain one.
AppendStream(const AFileName: string; const AStream: TStringStream)	Appends a string stream to a file. If the file does not exist, the file is created.
ColorToHTML(const AValue: TAlphaColor): string	Converts the TAlphaColor value to a HTML color string.
CreateFileFromResource(AFileName: string; AResourceName: string)	Creates a file from a resource.
Decode64Bytes(const AValue: string): TBytes	Decodes a base 64 string into an array of bytes.
Decode64String(const AValue: string): string	Decodes a base 64 string into a string.
Encode64Bytes(const AValue: TBytes): string	Encodes an array of bytes into a base 64 string.
Encode64String(const AValue: string): string	Encodes a string into a base 64 string.
ExtractPicture(const AValue: string): string	Extracts the picture data from a string that begins with #BEGINPIC# and ends with #ENDPIC# tags.
GetConcatenatedLogMessage(const AOutputInformation: TTMSLoggerOutputInformation; const AIndent: Boolean = False): string	Returns a concatenated log message based on the output information received from the logger. Optional parameters specify if indenting needs to be applied.
GetCurrentLangID: string	Returns the current language identifier.
GetDefaultOutputFileName: string	Returns the default output file name used inside an output handler that logs to a separate plain text or HTML file.

Name	Description
GetHTMLFormattedMessage(const AOutputInformation: TTMSLoggerOutputInformation; const AMode: TTMSLoggerHTMLOutputHandlerMode; const AApplyOutputParameters: Boolean; const ADocumentWrite: Boolean; const AEven: Boolean): string	Returns a HTML formatted message based on the output information received from the logger. Optional parameters specify the mode, if output parameters need to be applied such as the global color of the string, specifies whether document.write needs to be included and if the style applied to an element needs to include the even style.
GetIndent(const AIndent: Integer): string	Returns a string containing the AIndent amount of spaces.
GetProcessID: Cardinal	Returns the process id.
GetResourceStream(const AResourceName: string): TResourceStream	Returns a resource stream based on a resource name.
GetSystemInformation: string	Returns information on the operating system.
GetThreadID: Cardinal	Returns the thread id.
GetTickCountX: Integer	Returns the current tick count.
GetTimer(const AMode: TTMSLoggerTimerMode = lsmMilliseconds): Int64	Returns an elapsed value in ticks or milliseconds based on the mode parameters after a timer has been started with StartTimer.
GetUsedMemory: Cardinal	Returns the amount of memory the application is using.
HexStringToByteStream(const AValue: string; const ADigits: Integer = 2): TBytesStream	Returns a stream of bytes based on a hex string.
HexStrToBytes(const AValue: string; const ADigits: Integer = 2): TBytes	Returns an array of bytes based on a hex string.
IntToBinByte(const AValue: Byte; const ADecimals: Integer): string	Converts an integer to a byte with an optional amount of decimals.
IsTimerRunning	Returns a Boolean if the timer is still running after it was started with the StartTimer call.
LogToConsole(const AValue: string)	Logs a string value to the console window of the IDE or the log monitor / console of the device (iOS, Android, Mac OSX).

Name	Description
ReplaceTextInFile(AFileName, AText, AReplaceText: string)	Replaces a string inside a file.
StartTimer	Starts a timer, needs to be paired with StopTimer.
StopTimer(const AMode: TTMSLoggerTimerMode = lsmMilliseconds): Int64	Returns an elapsed value in ticks or milliseconds based on the mode parameters and stops the timer, needs to be paired with StartTimer.
StripHTML(const AValue: string): string	Strips HTML from a string.

When adding the unit `TMSLoggingCore`, there is a helper method TMSLog() to quickly log a value with optional format and level parameters. Below is a sample that demonstrates this.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  i := 100;
  TMSLog(i);
end;
```

Record and Class Helpers

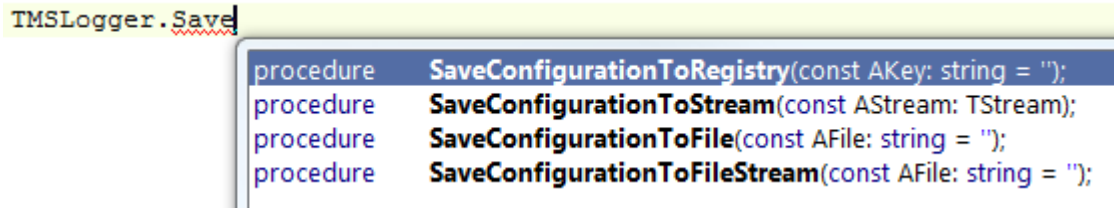
The core packages provides an additional unit which provides record and class helpers for a set of types available in Delphi. When adding the unit `TMSLoggingHelpers`, the default record / class helpers for the type you wish the use will be hidden. Unfortunately Delphi doesn't allow record / class helper inheritance, so the use of it is completely optional. Below is a sample what can be achieved when using this unit.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  fmt: string;
begin
  fmt := 'The value is {%g}';
  for I := 1 to 10 do
    I.LogInfoFormat(fmt);
end;
```

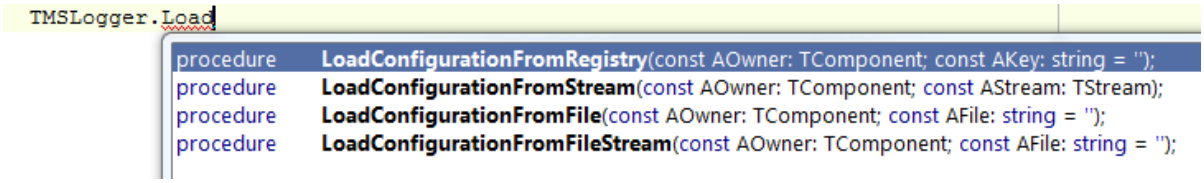
Note that the logging starts from the value itself, instead of passing it as a parameter to the logger log statements. This unit makes use of the custom logger instance retrieved with *TMSDefaultLogger*. The default logger instance retrieved with *TMSLogger* will be ignored, thus implying that by default, the *TMSDefaultLogger* will only log to the console output handler. When this technique is used, instead of the default *TMSLogger* functionality, the registration of output handlers need to be applied separately.

Persistence

The logger has the ability to save its configuration, registered output handlers and properties to a file, stream or registry (Windows only), so to save the logger instance, simply call one of the Save* methods.



To load, the Load* equivalent of the Save methods can be used. Please note that this will override any registered output handlers, or properties set.



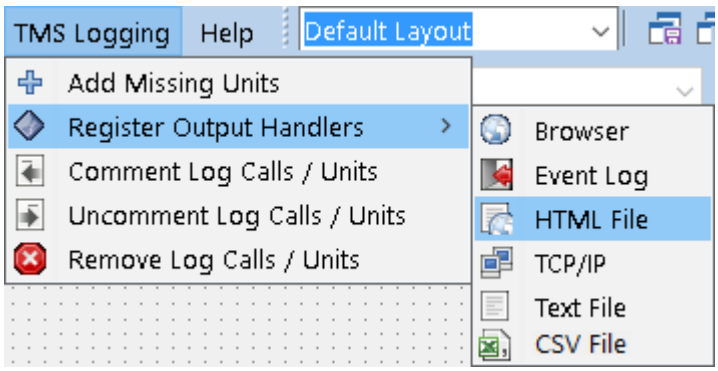
Each Save* and Load* call will automatically call the Save* and Load* calls on output handler instances. The *TTMSLoggerBaseOutputHandler* class provides a set of read and write calls for registry and ini file save / load instructions.

Initialization of the logger, such as the output handlers, output formats properties can be done once when saving the configuration. Simply loading the configuration again will automatically create any registered output handlers with their settings. For creation of the *TTMSLoggerBrowserOutputHandler* and *TTMSLoggerTCPOutputHandler*, the default constructor has an AOwner: TComponent parameter that needs to be set in order to successfully destroy the HTTP or TCP server instance. The parameter needs to be a form as demonstrated in the following sample, which registers a *TTMSLoggerBrowserOutputHandler*, saves the configuration to a stream and then reloads the configuration after unregistering all output handlers:

```
var
  ms: TMemoryStream;
begin
  TMSLogger.RegisterOutputHandlerClass(TTMSLoggerBrowserOutputHandler, [Self]);
  ms := TMemoryStream.Create;
  try
    TMSLogger.SaveConfigurationToStream(ms);
    TMSLogger.UnregisterAllOutputHandlers;
    ms.Position := 0;
    TMSLogger.LoadConfigurationFromStream(Self, ms);
  finally
    ms.Free;
  end;
end;
```

IDE Plugin

After installing TMS Logging through the automated installer, the IDE is updated with a "TMS Logging" helper menu that can be used to execute various operation when implementing logging in your application. Note that each action is only applied to the active source editor window, not application wide and the plugin is only supported in Delphi.



Add Missing Units (Keyboard shortcut ALT+M+A)

When adding logging to your application with the TMS Logging units, or copy and use the code snippets from this documentation, which possibly demonstrates the uses of an output handler, you might encounter compilation issues which indicates that there are missing units. This menu item will look into the active source file editor window and will automatically add missing units in order to compile your application. Optionally, to quickly add missing units, the shortcut ALT+M+A can be used.

Register Output Handlers

This menu option has a set of sub menu items that inserts registration code at the cursor position. Each output handler has a unique signature and the "Register Output Handler" menu item will help you setup the registration code needed to use the output handler.

Comment / Uncomment Log Calls / Units

This menu option will look for logger specific calls / units and will comment/uncomment them. This is designed to quickly eliminate any logger calls when you want to test your application without logging capabilities.

Remove Log Calls / Units

This menu option will look for logger specific calls / units and will remove them.

About

This documentation is for TMS Logging.

In this section:

What's New

Copyright Notice

Getting Support

Breaking Changes

What's New

Version 2.1 (Jun-2020)

- **New: Support for RAD Studio 10.4 Sydney.**
- **New: Support for Linux platform.**

Version 2.0 (May-2020)

- **New: Complete new installer, package structure and documentation format. This include breaking changes in the package files.**
- **New: Title property in boht HTML Output and Browser Output handlers allows for specifying the title of generated HTML page.**
- **New: RegisterManagedOutputHandler method allows adding an instance of an output handler to the logger and rely that it will be destroyed by it.**
- **Fixed:** HTML Output and Browser Output handlers not working correctly when the logged message had line breaks.

Version 1.5.1

- **Fixed:** High CPU usage when using TTMSLoggerBrowserOutputHandler.

Version 1.5

- **New: TTMSLoggerBaseOutputHandler.LogLevelFilters property allows filtering log messages on a per-outputhandler basis.**
- **New: TLoggingWindowsConsoleOutputHandler outputs log messages to the Windows console window.**
- **New: TLoggingSlackWebhookOutputHandler outputs log messages to a Slack channel through a incoming webhook.**
- **New: TLoggingMemoOutputHandler outputs log messges to a VCL TMemo control.**

Version 1.4.0.1

- **Improved:** Performance when fetching ip address for logging purposes.

Version 1.4

- **New: RAD Studio 10.3 Rio support.**

Version 1.3.0.2

- **Improved:** Performance when fetching ip address for logging purposes.

Version 1.3.0.1

- **Fixed:** Issue parsing HTML < and > signs in plain text.

Version 1.3

- **New: Custom log level mode and CustomLogLevel property at logger level.**

Version 1.1.0.1

- **Fixed: Version check for memory access in older devices.**

Version 1.1

- **New: Exception handling.**
- **New: Outputhandlers for Exceptionless.io and MyCloudData.**
- **New: TCP server component for handling multiple clients.**
- **New: TMSLoggingTCPOutputHandler client mode.**
- **Improved:** Active and Deactivate all outputhandlers with a single call.
- **Improved:** Allow creation of TTMSLoggerTCPOutputHandler and TTMSLoggerBrowserOutputHandler without form owner parameter (requires setting active = false in formclose).
- **Improved:** Public property Server for TTMSLoggerTCPOutputHandler and TTMSLoggerBrowserOutputHandler to manually configure multiple bindings.
- **Improved:** Automatic disconnect in TMSLoggingTCPClient.

Version 1.0

- **First Release.**

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.
- Make sure you have the latest version of the component(s).
- Search public TMS support channels (see below) to see if you question hasn't been already answer.

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.

Getting support

Visit our support page to learn about the channels for support:

https://www.tmssoftware.com/site/support.asp

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 2.0

TMS Logging packages have been restructured. The packages now use Libsuffix option so the dcp files are generated with the same name for all Delphi versions. Here is an overview of what's changed:

Before version 2.0, the package names were these:

```
TMSLoggingPkgCoreDXE<n>.dpk

TMSLoggingPkgFMXDXE<n>.dpk

TMSLoggingPkgVCLDXE<n>.dpk

TMSLoggingPkgDEDXE<n>.dpk
```

Where `<n>` is the Delphi version, being *7 = XE7* up to *12 = Delphi 10.3 Rio*.

However, the generated DCP had the same Delphi version number in name, e.g., `TMSLoggingPkgCoreDXE7.dcp` for Delphi XE7 and `TMSLoggingPkgCoreDXE12.dcp` for Delphi 10.3 Rio.

After this restructure, we took the opportunity to fix this and now DCP files have the same name regarding the Delphi version. The new package names are:

```
TMSLogging.dpk

TMSLoggingFMX.dpk

TMSLoggingVCL.dpk

dclTMSLogging.dpk
```

The names are respectives to the previous packages. DCP files are generated with same name, and only BPL files are generated with the suffix indicating the Delphi version. The suffix, however, is the same used by the IDE packages (numeric one indicating IDE version: 250, 260, etc.).