Fabasoft app.telemetry SDK

Fabasoft app.telemetry SDK 2024

Fabasoft

Contents

1 Introduction

1.1 Introduction

Fabasoft app.telemetry provides a Software-Telemetry SDK that enables applications to integrate into Fabasoft app.telemetry. Applications can integrate by registering themselves into Software-Telemetry and by providing information (events) used within Software-Telemetry log pools and sessions. These events help you to track the control flow through a distributed environment of your application.

1.2 Application Registration

To register your application and the modules in an accurate way you have four standard layers of registration information:

- `appName`: the application name
- `appId`: an application ID (an instance or major version of the base application)
- `appTierName`: an application tier or module (e.g. to separate backend from frontend logic)
- `appTierId`: the application tier ID (e.g. the service instance # 2)

Use the four standard layers of registration information to structure your services in an adequate way, so your application is represented meaningful in the infrastructure view of the Fabasoft app.telemetry client. The `appName` and `appId` represent the id of a top level service group. Inside you may group services (`appTierName`) by service-type (e.g. web-service vs. database-service) or service-context (e.g. put services for sales and accounting in different groups) and use the `appTierId` to distinguish between the particular services.

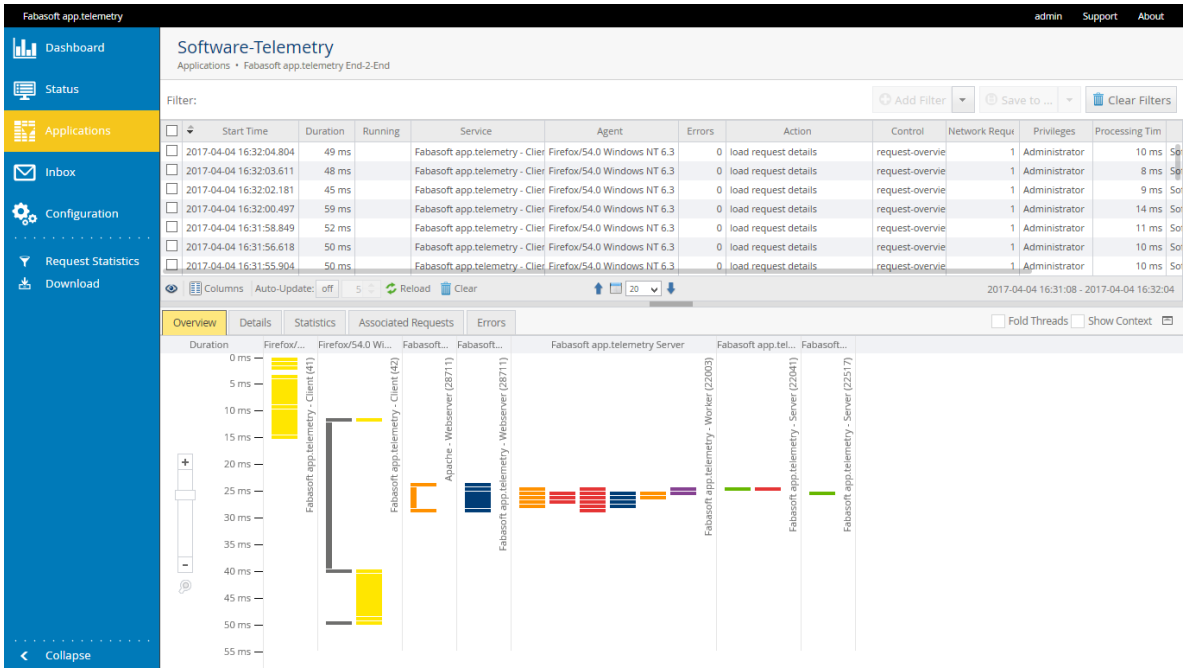
As of Fabasoft app.telemetry version 18.3 it is possible to register additional properties to allow greater flexibility of application registrations.

You can see each of these registered applications in a separate process column in the Software-Telemetry analysis view. Put the registration on a position in the source code of your application that runs once at startup of the application. The combination of all four parameters should be unique per Server/virtual Server (Cluster).

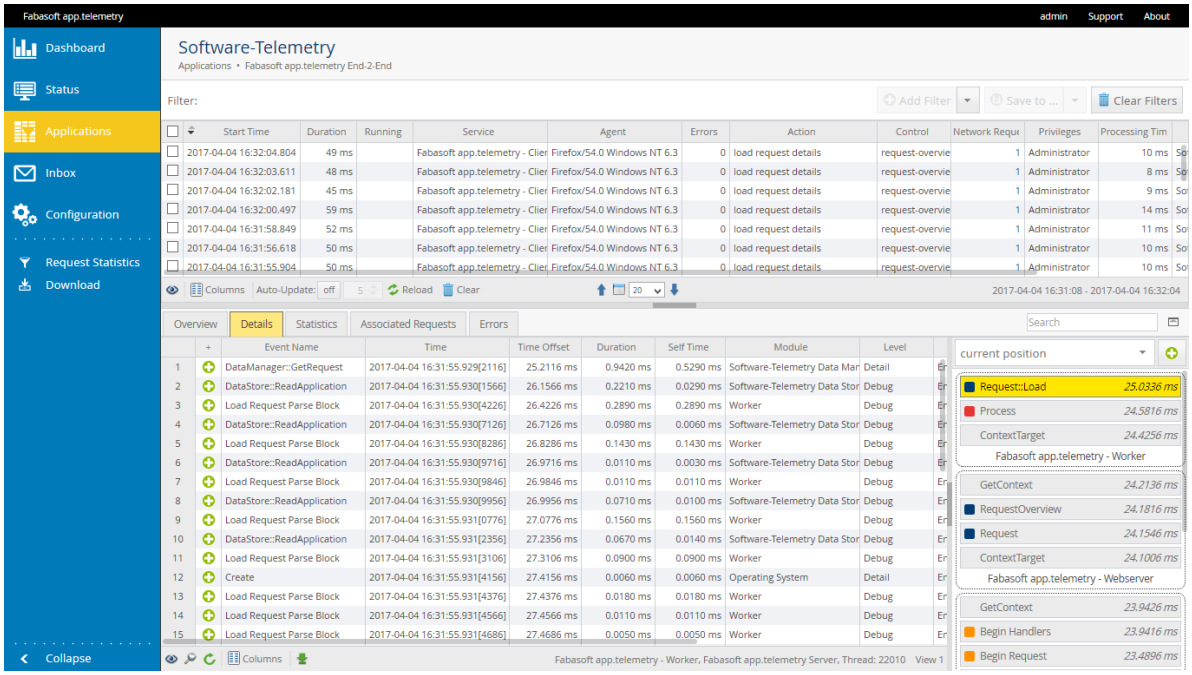
Note: Do not use random information or unpredictable information like process id or start time for registration parameters as this information is used to register services in the infrastructure and random information will result in multiple registered services.

1.3 Module Registration

For each module inside your application you may register a separate module. Each of these modules is drawn in a separate column inside the Software-Telemetry analysis view. For these modules you can register event descriptions. Put the registration on a position in the source code of your module that runs once at creation of the module.



This figure shows the overview of a selected request of the *Fabasoft app.telemetry Client* where you can navigate into the event details.



This figure shows the event details of a selected request of the Fabasoft app.telemetry Client where you can see time usage of any single event, parameter values and context handling. To navigate deeper into the details use the "plus"-icon. You can see the current position in your request in the call stack view on the right side where you can also navigate to other positions.

1.4 Counter Registration

As of Fabasoft app.telemetry Version 2018 UR 3 (18.3) applications can also register counters of various types to provide additional performance and status information that is independent of requests.

1.5 Standard Modules

With SDK version *2009 Summer Release* two standard module definitions and some general predefined events have been introduced. These modules define common instrumentation points for areas that are common to many projects.
Note: Do not use those predefined event-IDs listed below for your own events!

1.5.1 Operating System Module

The *Operating System* Module defines standard events for file system access, process execution and wait operations. These events should be used to instrument applications in a consistent way. To use this module, call the `RegisterModule`-function or create an object of class *APMModule* with a module name equal to *"Operating System"* using the constant APM_MODULE_NAME_OS. The event descriptions are statically defined in the app.telemetry server process and may therefore not be registered within the application.

Code Example

```
// register the module once
APMModuleHandle g_osmodule = APMRegisterModule(APM_MODULE_NAME_OS);
// …

// open a file
APMEventStr(g_osmodule, APM_EVENT_OS_OPEN, APM_EVENT_LEVEL_DETAIL,
APM_FLAG_ENTER, filename);
FILE *f = fopen(filename, "r");
APMEvent(g_osmodule, APM_EVENT_OS_OPEN, APM_EVENT_LEVEL_DETAIL,
APM_FLAG_LEAVE);
```

The constant names are prefixed with "`APM_`" for the C/C++ API (`APM_<constant>`) and in the other object-oriented languages they are members of the `APM` base class (`APM.<constant>`).

Name	Value	Eventname	Parameters	Description
`APM_MODULE_NAME_OS` `APM.MODULE_NAME_OS`	"Operating System"			
`APM_EVENT_OS_CREATE` `APM.EVENT_OS_CREATE`	`0x0000`	"Create"	"Name"	Create a file, stream or system object. Pass the resource name as parameter 1.
`APM_EVENT_OS_OPEN` `APM.EVENT_OS_OPEN`	`0x0001`	"Open"	"Name"	Open a file, stream or other system object. Pass the resource name as parameter 1.
`APM_EVENT_OS_DELETE`	`0x0002`	"Delete"	"Name"	Delete a file, stream or other system object. Pass

APM.EVENT_OS_DELETE				the resource name as parameter 1.
APM_EVENT_OS_READ APM.EVENT_OS_READ	0x0003	"Read"	"Name"	Read from a file or stream. Optionally pass the resource name or identifier as parameter 1.
APM_EVENT_OS_WRITE APM.EVENT_OS_WRITE	0x0004	"Write"	"Name"	Write to a file or stream. Optionally pass the resource name or identifier as parameter 1.
APM_EVENT_OS_CLOSE APM.EVENT_OS_CLOSE	0x0005	"Close"	"Name"	Close a file or stream. Optionally pass the resource name or identifier as parameter 1.
APM_EVENT_OS_FLUSH APM.EVENT_OS_FLUSH	0x0006	"Flush"	"Name"	Flush a file or disk. Pass the resource name as parameter 1.
APM_EVENT_OS_RENAME APM.EVENT_OS_RENAME	0x0007	"Rename"	"Source;Target"	Rename a file. Pass the source name as parameter 1 and the target name as parameter 2.
APM_EVENT_OS_COPY APM.EVENT_OS_COPY	0x0008	"Copy"	"Source;Target"	Copy a file. Pass the source name as parameter 1 and the target name as parameter 2.
APM_EVENT_OS_MOVE APM.EVENT_OS_MOVE	0x0009	"Move"	"Source;Target"	Move a file. Pass the source name as parameter 1 and the target name as parameter 2.
APM_EVENT_OS_EXECUTE APM.EVENT_OS_EXECUTE	0x0100	"Execute	"Command"	Execute a process. Pass the command line as parameter 1.
APM_EVENT_OS_WAIT APM.EVENT_OS_WAIT	0x0200	"Wait"	"Object"	Wait for a system object (e.g. semaphore). Optionally pass an object name identifying the system object.
APM_EVENT_OS_SLEEP APM.EVENT_OS_SLEEP	0x0201	"Sleep"		Suspend current process.
APM_EVENT_OS_CRITICAL SECTION_WAIT APM.EVENT_OS_CRITICAL SECTION_WAIT	0x0300	"CriticalSection:: Wait"	"Name"	Wait inside a critical section.

APM_EVENT_OS_CRITICAL SECTION_ENTER APM.EVENT_OS_CRITICAL SECTION_ENTER	0x0301	"CriticalSection:: Enter"	"Name"	Enter a critical section.
APM_EVENT_OS_CRITICAL SECTION_LEAVE APM.EVENT_OS_CRITICAL SECTION_LEAVE	0x0302	"CriticalSection:: Leave"	"Name"	Leave a critical section.

1.5.2 XMLHttpRequest

The `XMLHttpRequest` is defined to be used in an http client application to trace the synchronous or asynchronous events of a request.
The module has been used in the implementation of the SendXMLHttpRequest function in the JavaScript version of the SDK but may as well be used in your own application instrumentation.

The constant names are prefixed with "`APM_`" for the C/C++ API (`APM_<constant>`) and in the other object-oriented languages they are members of the `APM` base class (`APM.<constant>`).

Name	Value	Eventname	Parameters
APM_MODULE_NAME_XMLHTTP APM.MODULE_NAME_XMLHTTP	"XMLHttpRequest"		
APM_EVENT_XMLHTTP_SEND APM.EVENT_XMLHTTP_SEND	0	"Send"	
APM_EVENT_XMLHTTP_LOADING APM.EVENT_XMLHTTP_LOADING	1	"Loading"	
APM_EVENT_XMLHTTP_LOADED APM.EVENT_XMLHTTP_LOADED	2	"Loaded"	
APM_EVENT_XMLHTTP_INTERACTIVE APM.EVENT_XMLHTTP_INTERACTIVE	3	"Interactive"	
APM_EVENT_XMLHTTP_COMPLETE APM.EVENT_XMLHTTP_COMPLETE	4	"Completed"	"Status;Status Text"
APM_EVENT_XMLHTTP_PROCESSING APM.EVENT_XMLHTTP_PROCESSING	10	"Processing"	
APM_EVENT_XMLHTTP_PROCESSED APM.EVENT_XMLHTTP_PROCESSED	11	"Processed"	

1.5.3 HttpRequest

The `HttpRequest` Module is used to trace the parameters of http requests such as URL, method, status, header fields and message bodies.

The constant names are prefixed with "APM_" for the C/C++ API (APM_<constant>) and in the other object-oriented languages they are members of the APM base class (APM.<constant>).

Name	Value	Eventname	Parameters
APM_MODULE_NAME_HTTP APM.MODULE_NAME_HTTP	"HttpRequest"		
APM_EVENT_HTTP_URL APM.EVENT_HTTP_URL	2000	"URL"	"scheme;host;path;extra"
APM_EVENT_HTTP_METHOD APM.EVENT_HTTP_METHOD	2001	"Method"	"method"
APM_EVENT_HTTP_STATUS APM.EVENT_HTTP_STATUS	2002	"Status"	"code;message"
APM_EVENT_HTTP_AUTHENTICATION APM.EVENT_HTTP_AUTHENTICATION	2003	"Authentication"	"type;credential"
APM_EVENT_HTTP_PROXY APM.EVENT_HTTP_PROXY	2004	"Proxy"	"server;username"
APM_EVENT_HTTP_REQUEST_HEADER APM.EVENT_HTTP_REQUEST_HEADER	2010	"Request Header"	"header;value"
APM_EVENT_HTTP_REQUEST_CONTENTTYPE APM.EVENT_HTTP_REQUEST_CONTENTTYPE	2011	"Request Content Type"	"media-type;charset;boundary"
APM_EVENT_HTTP_REQUEST_CONTENTBODY APM.EVENT_HTTP_REQUEST_CONTENTBODY	2012	"Request Body Part"	
APM_EVENT_HTTP_RESPONSE_HEADER APM.EVENT_HTTP_RESPONSE_HEADER	2020	"Response Header"	"header;value"
APM_EVENT_HTTP_RESPONSE_CONTENTTYPE APM.EVENT_HTTP_RESPONSE_CONTENTTYPE	2021	"Response Content Type"	"media-type;charset;boundary"
APM_EVENT_HTTP_RESPONSE_CONTENTBODY APM.EVENT_HTTP_RESPONSE_CONTENTBODY	2022	"Response Body Part"	

1.6 Predefined Events

The SDK already provides a set of predefined events used for general purpose.

Note: Do not use any of these predefined event-IDs for your own event declarations!

1.6.1 Error/Trace Events

The constant names are prefixed with "`APM_`" for the C/C++ API (`APM_<constant>`) and in the other object-oriented languages they are members of the `APM` base class (`APM.<constant>`).

Constant Name	Value
`APM_EVENT_ERROR` `APM.EVENT_ERROR`	`0x10001000`
`APM_EVENT_WARNING` `APM.EVENT_WARNING`	`0x10001001`
`APM_EVENT_INFO` `APM.EVENT_INFO`	`0x10001002`
`APM_EVENT_TRACE` `APM.EVENT_TRACE`	`0x10001003`

Use these standard events to mark error, warning or info conditions in your application like in the following example:

Code Example

```
APMEventStr(anymodule, APM_EVENT_ERROR, APM_EVENT_LEVEL_NORMAL,
APM_FLAG_ERROR, "My error message”);
APMEventStr(anymodule, APM_EVENT_WARNING, APM_EVENT_LEVEL_NORMAL,
APM_FLAG_WARNING, "My warning message”);
APMEventStr(anymodule, APM_EVENT_INFO, APM_EVENT_LEVEL_NORMAL,
APM_FLAG_NONE, "My info message”);
APMEventStr(anymodule, APM_EVENT_TRACE, APM_EVENT_LEVEL_DETAIL,
APM_FLAG_ERROR, "My trace message”);
```

1.6.2 Events for Defining Sequence Hierarchies

The Sequence functionality has been deprecated and has been removed in Version 2015. Register your own events and log columns instead.

1.7 Predefined Basic Constants

1.7.1 Flags

Flags are used to specify the type of an event and are passed as argument to every event call.

The constant names are prefixed with "`APM_`" for the C/C++ API (`APM_<constant>`) and in the other object-oriented languages they are members of the `APM` base class (`APM.<constant>`).

Name	Value	Description
APM_FLAG_NONE APM.FLAG_NONE	0x00	normal event
APM_FLAG_ENTER APM.FLAG_ENTER	0x01	indicates a starting point
APM_FLAG_LEAVE APM.FLAG_LEAVE	0x02	indicates an ending point
APM_FLAG_WAIT APM.FLAG_WAIT	0x04	indicates a wait condition
APM_FLAG_WARNING APM.FLAG_WARNING	0x08	mark event as warning
APM_FLAG_ERROR APM.FLAG_ERROR	0x10	mark event as error
APM_FLAG_PROCESS_INFO APM.FLAG_PROCESS_INFO	0x20	force writing *SessionInfo* just before the event

1.7.2 Log Level

An event will only be logged if the event-level is lower or equal to the value selected in the session or log-definition.

The constant names are prefixed with "`APM_`" for the C/C++ API (`APM_<constant>`) and in the other object-oriented languages they are members of the `APM` base class (`APM.<constant>`).

Name	Value	Description
APM_EVENT_LEVEL_LOG APM.EVENT_LEVEL_LOG	00	for events that should always be included and for logging parameters for the request overview
APM_EVENT_LEVEL_IPC APM.EVENT_LEVEL_IPC	10	INTERNAL value
APM_EVENT_LEVEL_NORMAL APM.EVENT_LEVEL_NORMAL	50	for general events for normal recording sessions
APM_EVENT_LEVEL_DETAIL APM.EVENT_LEVEL_DETAIL	60	for detail level events for detailed session analysis
APM_EVENT_LEVEL_DEBUG APM.EVENT_LEVEL_DEBUG	70	event with debugging information - highest level of precision with huge data amount and performance impact

1.7.3 Application Properties

Application Properties are static and unchanging properties used in addition to the four standard registration parameters when creating service objects in the app.telemetry infrastructure.

The constant names are prefixed with "`APM_`" for the C/C++ API (`APM_<constant>`) and in the other object-oriented languages they are members of the `APM` base class (`APM.<constant>`).

Name	Description
`APM_APP_PROPERTY_INSTANCE_GUID_KEY` `APM.PROPERTY_INSTANCE_GUID_KEY`	Setting the GUID (Globally Unique Identifier) of an application enables app.telemetry to track the same application instance among agents without creating a new service on every agent the application is ever launched on. This can be used as an alternative for the built in clustering support.
`APM_APP_PROPERTY_INSTANCE_GUID_NAME` `APM.PROPERTY_INSTANCE_GUID_NAME`	Display name for the instance GUID property. (Setting a custom name for this property is not supported).

1.7.4 Application Values

Application values can provide additional, changeable (very low change frequency) information about an application or it's runtime environment.

The constant names are prefixed with "`APM_`" for the C/C++ API (`APM_<constant>`) and in the other object-oriented languages they are members of the `APM` base class (`APM.<constant>`).

Name	Description
`APM_APP_VALUE_VERSION_KEY` `APM.VALUE_VERSION_KEY`	Key for the application version value.
`APM_APP_VALUE_VERSION_NAME` `APM.VALUE_VERSION_NAME`	Display name for the application version value. (Setting a custom name for this value is not supported).
`APM_APP_VALUE_INSTANCE_NAME_KEY` `APM.VALUE_INSTANCE_NAME_KEY`	Key for the instance name property used to provide a nice display name for services identified via the instance GUID property.
`APM_APP_VALUE_INSTANCE_NAME_NAME` `APM.VALUE_INSTANCE_NAME_NAME`	Display name for the instance name value. (Setting a custom name for this value is not supported).

1.8 Request and Context Handling

After your application and all defined modules are registered you can start with the instrumentation of your program sequence.

When the request enters your application you have to create a Software-Telemetry context with a specific filter value. This filter value should be one of those you have registered for any module of

this process with the `RegisterFilterValue`-method. By calling the `CreateContext` function with a filter value matching the filter value of a started Software-Telemetry session recording is enabled.

This context information is used by the analysis components to track the control flow through the distributed environment. Then you can subsequently fire events which are tracked by the Software-Telemetry. After firing the last event you have to call the method `ReleaseContext` to finish recording events and to tell app.telemetry that processing has finished.

1.8.1 Passing Context

If processing in your application involves other services (thread/process/module) you may track control and data flow between services by passing the Software-Telemetry context between the services.

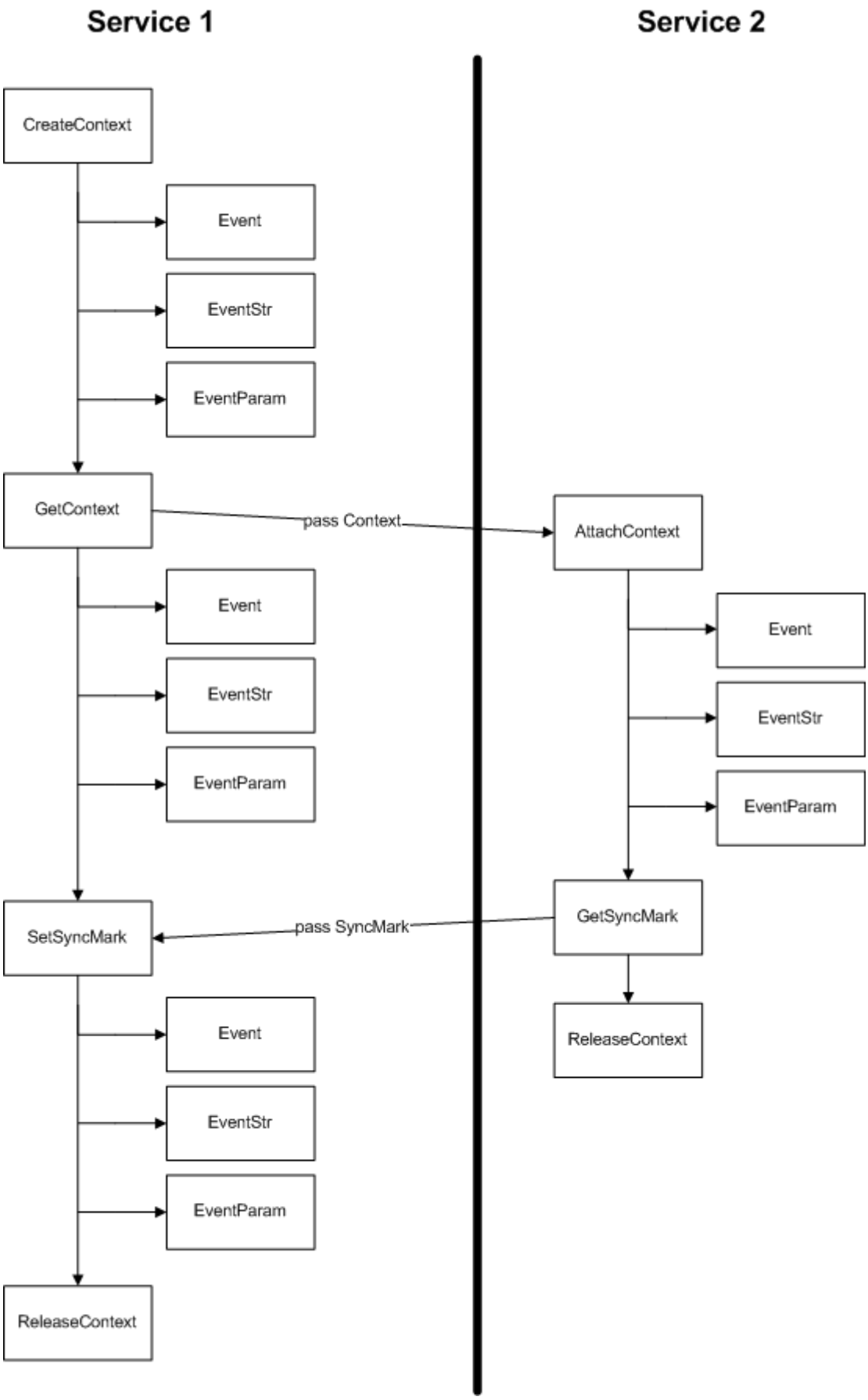
With the `GetContext`-method you acquire a context token. Then you pass the token with your request data to the other service being invoked (for example by transmitting the context token over the network stream).

In the second service you use the token to attach to this context using the `AttachContext`-method. By doing this Software-Telemetry recording is being enabled in the called service and data will be displayed accordingly in the analysis view.

To pass the context token to an HTTP endpoint it must be set as a header with the name "`x-apm-telemetry-context`" and the value must be encoded using base64, this allows intermediaries such as load-balancers or reverse proxies to insert themselves in between the communicating parties to provide a more complete picture of the request flow. Instrumented HTTP endpoints must respond with a sync mark token as an http header with the name "`x-apm-telemetry-syncmark`" that is also base64 encoded for the same reason.

1.8.2 Synchronizing Timeline with SyncMarks

To get a correct timeline if the first service depends on the output of the second you have to use synchronization markers. Inside the second service you have to acquire a synchronization marker using the function `GetSyncMark`. After doing this you have to pass the synchronization marker together with the response back to the first service. Then you call the method `SetSyncMark` inside the first service. In the following figure you can see an illustration of the program flow.


Service 1
Service 2
CreateContext
Event
EventStr
EventParam
GetContext
pass Context
AttachContext
Event
EventStr
EventParam
Event
EventStr
EventParam
SetSyncMark
pass SyncMark
GetSyncMark
Event
EventStr
EventParam
ReleaseContext
ReleaseContext

1.9 Log Definitions

Each event that is recorded by any Fabasoft app.telemetry agent (when the instrumented application fires these events) is sent to the Fabasoft app.telemetry server, which is the responsible service to process and analyze these events.

The Fabasoft app.telemetry server requires a definition (analyzer structure) for interpreting and analyzing the Software-Telemetry events.

The developer who instruments the application is responsible to provide a log definition for his instrumentation points. A log definition is an *XML* file containing a description of events, their parameters with the data type and settings how to persist these events in a database for statistical analysis.

1.9.1 Application Specific Log Definitions (Analyzer Structure)

The app.telemetry Software-Telemetry analyzer module on the Fabasoft app.telemetry server requires at least one valid log definition to be defined. Otherwise no telemetry data will be processed.

The log definitions are normally delivered by the application vendor (by the developer who instrumented the application by means of Software-Telemetry SDK). These log definitions must be imported into the infrastructure model via the Fabasoft app.telemetry client by means of importing an *XML* file in the client interface in the edit mode.

1.9.2 Log Definition Syntax

Log definitions must have the following *XML* format:

Code Example

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<APMLogAnalyzer>
  <APMLogAnalyzerEntry  module="My Module Name"  eventid="12345"
      paramid="1"  name="Event Param Name" type="2"
      tablename="DB Table Name"  tabletype="1"
      columndatatype="2"  [columnlength="200"]
      [flags="0"]  [format="formatstring"]  [block="eventname"]/>
  <APMLogAnalyzerEntry eventid="12345" ... />
  ...
</APMLogAnalyzer>
```

The attributes of an entry are defined as follows:

- `eventid`: as defined in module implementation
- `module`: name of module as defined in module implementation
- `paramid`: ID of parameter `<1..n>` - starting at index 1 and the max value is the number of available parameters for this event. For block entries the `paramid` is set to `0`, because block entries are virtual calculated values.
- `name`: the desired name for this log entry (will be used to display parameter column header and to persist in DB). This name must not contain any special characters and should not contain spaces – must be a valid database column name
- `type`: `<0..6>` … data type of entry definition – nearly the same as `columndatatype`
  - `0` … block start event type (has no data type)
  - `1..6`: same as `columndatatype` (both type columns have to be equal for normal entries)

- `tablename`: Database table name where to store this entry. This name must not contain any special characters and spaces - must be a valid database table name.
- `tabletype`: `<1|2>` ... type of database table where to store entry
  - `1` ... defines the base table storing all relevant meta information for the request. (There should only be one table of `tabletype 1` in a single `APMLogAnalyzer` definition, because all base table columns are stored for this `tabletype`)
  - `2` ... additional table for data that may occur more than once within a request (for example: database SQL query).
- `columndatatype`: `<1..6>` ...data type how to persist this entry in the DB
  - `1` ... *Int32*: 32 bit integer number
  - `2` ... *Int64*: 64 bit integer number
  - `3` ... *String*: textual value – if this data-type is set, the attribute `columnlength` must be set too.
  - `4` ... *Duration*: used for calculated duration between two events (when block is set) – unit in database = [100ns] (that means value 10000 in DB = 1ms); the Fabasoft app.telemetry Client displays the duration already formatted in milliseconds.
  - `5` ... *Timestamp*: 64 bit long value used for storing date/time information
  - `6` ... *Double*: 64 bit floating point number
- `columnlength`: length for database column definition (number of characters) – only required for data type 3 (string)
- `format`: optional application specific format definition
  for example: `fsc:address`, `fsc:reference`, `fsc:name`
- `flags`: bit-mask of additional flags:
  Specify the sum of the flags as a decimal value in the flags attribute.
  - `0x0004` ... start of block
  - `0x0008` ... end of block
  - calculation type (only one can be selected)
    - `0x0000` ... unique entry (first occurrence)
    - `0x0010` ... entry could occur multiple times – each event occurrence will be persisted (if used in combination with *tabletype* 2)
    - `0x0020` ... unique entry (last occurrence) - event will be overwritten each time a new instance of the same event is fired
    - `0x0030` … count (counts number of event occurrences)
    - `0x0040` … add (multiple values are added)
  - `0x0100` … the column is a dimension (valid only for integer and string type columns)
  - `0x0200` … the column is a measure (valid only for integer and duration type columns)
  - `0x0400` … the measure column is null-able, so a count column of not null values is calculated (valid only for measure columns)
  - `0x0800` … calculate minimum (valid only for measure columns)
  - `0x1000` … calculate maximum (valid only for measure columns)
  - generic column (based on calculated other column) ... all these generic column definitions require the *parent*-attribute to define the base column ("name") containing the base values for the new calculated column:
    - `0x10000` ... categorize value of other column: requires *calculation* and *format* attributes:
    - `calculation` ... defines value split points

        - `format` ... to define a textual representation of the classified values
        - example to split the request duration into classes: `parent="duration" calculation="10000000;50000000;150000000" format="enum:1:0-1s;2:1-5s;3:5-15s;4:>15s"`
        - `0x20000` ... split value of other column (with regular expression): requires *calculation* attribute
        - `calculation` ... defines the regular expression for splitting the base value into a new resulting value
        - example to split up the only the URL path from a full URL containing additional URL arguments: `parent="Page URL (referer)" calculation="[\w]+://[^/]*([^\?]+)" paramid="1"` ... enclose the desired text part in the regex with grouping braces - define the desired group match number with the *paramid*-attribute (first/only 1 matching group ... paramid="1").
        - `0x40000` ... application property (since Version 21.1)
        - `parent` ... the name of the application property or registration parameter (apm:appName, apm:appId, apm:appTierName, apm:appTierId, apm:hostname)
        - example to get a column filled with the application tier id: `parent="apm:appTierId"`.
    - Format options mask `0xF00000`
        - `0x100000` ... value is shown as IP address
        - `0x200000` ... value is shown as URL
        - `0x300000` … text values of agent or service column are stored on database to provide texts when agent objects are deleted
    - `0x1000000` … the column is anonymized (valid only for integer and string type columns)
    - `0x2000000` … the column is used as quick filter option in the telemetry and research view
- `block`: optional definition of block of events – used to block two events to one together (to measure for example the time between)
    - use flag 4 as start event and flag 8 as end event and data type 4 (duration)

Example: Log Definition

```
<?xml version=”1.0” encoding=”UTF-8” standalone=”no” ?>
<APMLogAnalyzer>
  <APMLogAnalyzerEntry eventid=”1105” module=”HTTP Server” paramid=”1”
    name=”recvbytes” type=”2” tablename=”request” tabletype=”1”
    columndatatype=”2” />
  <APMLogAnalyzerEntry eventid=”1109” module=”HTTP Server” paramid=”1”
    name=”arguments” type=”3” tablename=”request” tabletype=”1”
    columndatatype=”3” columnlength=”200” />
  <APMLogAnalyzerEntry eventid=”1100” module=”Virtual Application”
    paramid=”1” name=”srcappview” type=”2” tablename=”request”
    tabletype=”1” columndatatype=”2” format=”fsc:reference” />
  <APMLogAnalyzerEntry eventid=”1101” module=”Virtual Application”
    paramid=”1” name=”srcobject” type=”2” tablename=”request”
    tabletype=”1” columndatatype=”2” format=”fsc:address” />
  <APMLogAnalyzerEntry eventid=”1106” module=”Virtual Application”
    paramid=”1” name=”recvbytestx” type=”2” tablename=”request”
    tabletype=”1” columndatatype=”2” />
```

```
  <APMLogAnalyzerEntry eventid="2003" module="Fabasoft Components Kernel"
    paramid="1" name="query" type="3" tablename="query" tabletype="2"
    columndatatype="3" columnlength="4000" flags="16"/>

  <APMLogAnalyzerEntry eventid="352" module="WebDAV" paramid="1"
    name="method" type="3" tablename="request" tabletype="1"
    columndatatype="3" columnlength="20" flags="32"/>

  <APMLogAnalyzerEntry eventid="104" module="Virtual Application"
    paramid="0" type="0" block="readvalues" flags="4" />

  <APMLogAnalyzerEntry eventid="104" module="Virtual Application"
    paramid="0" name="readtime" type="4" tablename="request"
    tabletype="1" columndatatype="4" block="readvalues" flags="8"/>

</APMLogAnalyzer>
```

Explanation of example (above): this definition defines several different events of different types.

- *Query* (ID: `2003`):
  - The SQL database query is recorded as event with ID "`2003`" in the application in the module "Fabasoft Components Kernel".
  - The parameter will be named as "`query`" in the Software-Telemetry logs and in the database table "`query`".
  - The database table type "`2`" defines to use an extra table which does only contain the query and a reference to the basic entry in the common table.
  - The data type of this entry is `3`, which defines that the query is recorded as a string. The reserved field size in the database table is set to 4000 characters.
  - The flags value `16` means that each occurrence of this event is persisted in the extra database table "`query`".
- *Readtime* (ID: `104`):
  - The event with ID "`104`" of the module "Virtual Application" is handled as specific block, which means that this event is recorded as duration between two corresponding events.
  - For block entries the "`paramid`" is set to `0`, because these entries are calculated entries with no direct value passed from the event.
  - The first entry defines the start of the block with `type="0"` (irrelevant for start entry – defined in end entry) flags="4" and the block was named "`readvalues`".
  - The end entry (`flags="8"`) also requires to have the same block name, so that the server knows that these two block definitions belong together. The end entry defines the final entry name "`readtime`" and the data type "`4`" (duration). This entry will be persisted in the base table "`request`" (type `1`).

1.10 Defining a Sequence Hierarchy

The Sequence column functionality has been deprecated and has been removed in Version 2015. Register your own events and log columns instead.

1.11 Custom Telemetry Module Registration via Package

Since Fabasoft app.telemetry 2014 Summer Release it is possible to extend the module and event registration information done during instrumenting an application in the code by means of providing a custom "wizard" package containing special registration information:

- `<app>cfg.zip`

- `package.xml`: only containing a packageid for your application and a version of the current package file

Package.xml

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>

<Package name="Fabasoft Folio Integration Package" description="Fabasoft
Folio Integration Package" packageid="foliocfg" version="14.2.0.6"
author="Fabasoft R&amp;amp;D GmbH" icon="">

 </Package>
```

- `module_registration.xml`: this file provides custom color definitions for any of your application modules (if not defined a fallback color is used) and custom improved display names for special telemetry events and format definitions event parameters.

module_registration.xml

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<Package>
  <Module name="Fabasoft Folio Kernel" color="rgb(32,188,102)">

    <Follower name="Operating System"/>
    <Follower name="Database Library"/>
    <Follower name="Network Request"/>

    <Application applname="Fabasoft Folio" appltiername="Webservice"/>
    <Application applname="Fabasoft Cloud" appltiername="Webservice"/>


    <Event id="2002" name="Method::Call" displayname="{action}"
format="object{fsc:name};objclass{fsc:reference};action{fsc:reference}"/>

  </Module>

  <Module name="Virtual Application" color="rgb(211,95,0)">

    <Follower name="WebDAV"/>
    <Follower name="CardDAV"/>
    <Follower name="Virtual Application Controls"/>

    <Application applname="Fabasoft Folio" appltiername="Webservice"/>
    <Application applname="Fabasoft Cloud" appltiername="Webservice"/>

    <Event id="106" name="ProcessUseCase" displayname="{usecase}"
format="usecase{fsc:reference};entertype"/>
    <Event id="217" name="GenerateView"/>
    <Event id="218" name="ExploreNode" displayname="ExploreNode {node}"
format="node{fsc:name}"/>
    <Event id="219" name="GenerateFatalError" displayname="FatalError
{error}" format="error"/>

  </Module>
</Package>
```

The module name must match with the registered application module in the telemetry module registration and you can define a specific color that should be used for that module everywhere in the app.telemetry client.

The rule set can be applied for multiple application instances running with different application names or ids just by specifying a list of valid application registration mapping (e.g. the application

can register as “Fabasoft Folio” or “Fabasoft Cloud” and the above example will apply for requests from both applications.

You can now assign improved display names for any of your telemetry events identified inside the defined module with the event id. **Note**: every listed event in this package file will overwrite the original event registration. The following properties can be defined/overwritten with an improved value:

- `name`: the original event name or an improved one.
- `displayname`: can be a simple text string extended with dynamic values fetched from the event parameters referenced by the parameter {name} defined in the format attribute.
- `format`: for naming the parameter values and for special formatting of raw values (like `fsc:reference` for COO-addresses) - see 1.9.2 “*Log Definition Syntax*”

The new displayname for telemetry events is on the one hand available as additional table column in the request details table and on the other hand it is used as default name in the request call stack (the tooltip will show the name and the displayname property as well as some additional information).

Additional to the event definitions the modules can also define a specific ordering of the modules in the request overview chart by means of listing all possible followers of every module.

2 C/C++ API Reference

The Fabasoft app.telemetry Software-Telemetry C/C++ API allows you to instrument any C/C++ application with instrumentation points to see what's going on in complex applications.

2.1 Constants(C/C++)

This chapter lists the available constants to be used for instrumenting your C/C++ application.

2.1.1 Flags (C/C++)

Flags are used to specify the type of an event and are passed as argument to every event call. They are predefined by the API with the data type `UInt8` and have the following possible values:

- `APM_FLAG_NONE`
- `APM_FLAG_ENTER`
- `APM_FLAG_LEAVE`
- `APM_FLAG_WAIT`
- `APM_FLAG_WARNING`
- `APM_FLAG_ERROR`
- `APM_FLAG_PROCESS_INFO`

For more details about the flags and there meaning see the general chapter 1.7.1 “*Flags*”.

2.1.2 Log Level (C/C++)

An event will only be logged if the event-level is lower or equal to the value selected in the session or log-definition. The log levels are predefined by the API with the data type `UInt8` and have the following possible values:

- `APM_EVENT_LEVEL_LOG`
- `APM_EVENT_LEVEL_IPC` (only internal)
- `APM_EVENT_LEVEL_NORMAL`
- `APM_EVENT_LEVEL_DETAIL`
- `APM_EVENT_LEVEL_DEBUG`

For more details about the log levels and there meaning see the general chapter 1.7.2 “*Log Level*”.

2.1.3 Parameter Types (C/C++)

Parameter types provide type information of parameters in a call to `APMEventArgs` (see 2.5.2 “*Method APMEventArgs (C/C++)*”)

Name	Value	Description
`APM_PARAMETERTYPE_INT32`	`1`	parameter is of type UInt32
`APM_PARAMETERTYPE_INT64`	`2`	parameter is of type UInt64
`APM_PARAMETERTYPE_STRING`	`3`	parameter is an utf-8 encoded string(0 terminated)

2.1.4 Predefined Modules (C/C++)

For a list of available standard modules with a set of predefined events see chapter 1.5 “*Standard Modules”.*

2.1.5 Predefined Events (C/C++)

The SDK already provides a set of predefined events used for general purpose. For a full listing of all available predefined events see the generic chapter 1.6 "*Predefined Events*".

Error/Trace Events:

Use these standard events to mark error, warning or info conditions in your application like in the following example:

- `APM_EVENT_ERROR`
- `APM_EVENT_WARNING`
- `APM_EVENT_INFO`
- `APM_EVENT_TRACE`

Code Example

```
APMEventStr(anymodule, APM_EVENT_ERROR, APM_EVENT_LEVEL_NORMAL,
APM_FLAG_ERROR, "My error message”);

APMEventStr(anymodule, APM_EVENT_WARNING, APM_EVENT_LEVEL_NORMAL,
APM_FLAG_WARNING, "My warning message”);

APMEventStr(anymodule, APM_EVENT_INFO, APM_EVENT_LEVEL_NORMAL,
APM_FLAG_NONE, "My info message”);

APMEventStr(anymodule, APM_EVENT_TRACE, APM_EVENT_LEVEL_DETAIL,
APM_FLAG_ERROR, "My trace message”);
```

2.1.6 Predefined Application Property Keys and Names (C/C++)

Key	Name	Purpose
`APM_APP_PROPERTY_INSTANCE_GUID_KEY`	`APM_APP_PROPERTY_INSTANCE_GUID_NAME`	Instance ID

2.1.7 Predefined Application Value Keys and Names (C/C++)

Key	Name	Purpose
`APM_APP_VALUE_VERSION_KEY`	`APM_APP_VALUE_VERSION_NAME`	Application Version
`APM_APP_VALUE_INSTANCE_NAME_KEY`	`APM_APP_VALUE_INSTANCE_NAME_NAME`	Instance Name

2.1.8 Counter Attributes (C/C++)

Counter attributes define behavior, aggregation, measurement, representation and labels of registered counters. Some of the attributes can be changed but most must be defined during the initial counter registration.

Name	Description	Type
APM_COUNTER_ATTRIBUTE_NONE	End marker for attribute definition lists.	-
APM_COUNTER_ATTRIBUTE_GROUP_DISPLAY_NAME	Group display name of the counter.	Pointer
APM_COUNTER_ATTRIBUTE_DISPLAY_NAME	Display name of the counter.	Pointer
APM_COUNTER_ATTRIBUTE_INSTANCE_NAME	Name of the counter instance.	Pointer
APM_COUNTER_ATTRIBUTE_INSTANCE_DISPLAY_NAME	Instance display name of the counter.	Pointer
APM_COUNTER_ATTRIBUTE_DATA_SIZE	Length of the memory buffer for String values.	Number
APM_COUNTER_ATTRIBUTE_CALLBACK	APMCounterCallback structure address.	Pointer
APM_COUNTER_ATTRIBUTE_MEASUREMENT_INTERVAL	Interval (in Seconds) of automatic counter recording.	Number
APM_COUNTER_ATTRIBUTE_AGGREGATION_FUNCTION_NAME	See APM_COUNTER_AGGREGATION_FUNCTION_* constants.	Pointer
APM_COUNTER_ATTRIBUTE_APPLICATION	APMApplicationHandle of the application that owns the counter.	Number
APM_COUNTER_ATTRIBUTE_BASE	Base for fractional counter values.	Number
APM_COUNTER_ATTRIBUTE_WARNING_LIMIT	Warning limit for service check, related to the service check property “warning”, given as string.	Pointer
APM_COUNTER_ATTRIBUTE_ERROR_LIMIT	Critical limit for service check, related to the service check property “error”, given as string.	Pointer

Remarks:

The two attributes `APM_COUNTER_ATTRIBUTE_WARNING_LIMIT` and `APM_COUNTER_ATTRIBUTE_ERROR_LIMIT` are available with Version 2021 UR 1. Use them to automatically configure a corresponding service check with „warning“ and „error“ property as given, like in the following examples:

Attribute	Value	Service Check
APM_COUNTER_ATTRIBUTE_WARNING_LIMIT	“80”	The status of the service check is “Warning” if the value of the counter is above 80.
APM_COUNTER_ATTRIBUTE_ERROR_LIMIT	“::90”	The status of the service check is “Critical” if the value of the counter is below 90.
APM_COUNTER_ATTRIBUTE_WARNING_LIMIT	“10::20[15]”	The status of the service check is “Warning” if the value of the counter is above 10 and below 20 for a duration of 15

		seconds.
APM_COUNTER_ATTRIBUTE_ERROR_LIMIT	“error”	The status of the service check is “Critical” if the text of the counter matches “error” (for Counter Type APM_COUNTER_DATATYPE_TEXT).

2.1.9 Counter Types (C/C++)

Name	Description
APM_COUNTER_TYPE_RAW	Raw counter without special interpretation.
APM_COUNTER_TYPE_COUNTER	Per second values.
APM_COUNTER_TYPE_FRACTION	Percent value.

2.1.10 Counter Datatypes (C/C++)

Name	Description
APM_COUNTER_DATATYPE_UINT32	32-Bit Unsigned Integer
APM_COUNTER_DATATYPE_UINT64	64-Bit Unsigned Integer
APM_COUNTER_DATATYPE_TEXT	UTF-8 Encoded Text

2.1.11 Aggregation Function Names (C/C++)

Name	Description
APM_COUNTER_AGGREGATION_FUNCTION_SUM	Sum
APM_COUNTER_AGGREGATION_FUNCTION_AVG	Average
APM_COUNTER_AGGREGATION_FUNCTION_MAX	Maximum
APM_COUNTER_AGGREGATION_FUNCTION_MIN	Minimum

2.2 Data types (C/C++)

2.2.1 APMApplicationHandle (C/C++)

APMApplicationHandle is used to identify an application registered in your process by calling APMRegisterApplication (see 2.3.1 “*Method APMRegisterApplication (C/C++)*”). The handle is only valid in the same process until you call APMUnregisterApplication (see 2.3.2 “*Method APMUnregisterApplication (C/C++)*”)

Type Declaration

```
typedef UInt32 APMApplicationHandle;
```

2.2.2 APMModuleHandle (C/C++)

`APMModuleHandle` is used to identify a module registered in your process by calling `APMRegisterModule` (see 2.3.5 "*Method APMRegisterModule (C/C++)*"). The handle is valid only in the same process until you call `APMUnregisterModule` (see 2.3.6 "*Method APMUnregisterModule (C/C++)*").

Type Declaration
`typedef UInt32 APMModuleHandle;`

2.2.3 APMCounterHandle (C/C++)

`APMCounterHandle` is used to identify a counter or counter instance registered using `APMRegisterCounter` (see 2.3.9 "*Method APMRegisterCounter (C/C++)*").

Type Declaration
`typedef uint32_t APMCounterHandle;`

2.2.4 APMCounterAttributeType (C/C++)

`APMCounterAttributeType` is used to identify which attribute is described by the encompassing `APMCounterAttribute` structure. See 2.1.8 "*Counter Attributes (C/C++)*" for valid values.

Type Declaration
`typedef uint32_t APMCounterAttributeType;`

2.2.5 APMCounterType (C/C++)

`APMCounterType` specifies the type of the counter registered using `APMRegisterCounter` (see 2.3.9 "*Method APMRegisterCounter (C/C++)*"). See 2.1.9 *"Counter Types (C/C++)"*2.1.9 for a list of valid values.

Type Declaration
`typedef uint8_t APMCounterType;`

2.2.6 APMCounterDataType (C/C++)

`APMCounterDataType` specifies the data-type of the counter registered using `APMRegisterCounter` (see 2.3.9 "*Method APMRegisterCounter (C/C++)*"). See 2.1.10 "*Counter Datatypes (C/C++)*" for a list of valid values.

Type Declaration
`typedef uint8_t APMCounterDataType;`

2.2.7 APMCounterAttribute (C/C++)

`APMCounterAttribute` is used to pass attributes to the registration function `APMRegisterCounter` (see 2.3.9 "*Method APMRegisterCounter (C/C++)*").

Type Declaration

```
typedef struct {
  APMCounterAttributeType attributeType;
  int64_t value;
  const void *pointer;
} APMCounterAttribute;
```

2.2.8 APMCounterCallback (C/C++)

`APMCounterCallback` is used to pass a counter callback with optional custom data to the `APMRegisterCounter` (see 2.3.9 "*Method APMRegisterCounter (C/C++)*") function via the attributes as an `APM_COUNTER_ATTRIBUTE_CALLBACK`.

Type Declaration

```
typedef struct {
  void (*function)(void*);
  void *data;
} APMCounterCallback;
```

2.2.9 APMEventId (C/C++)

`APMEventId` is used to identify an event. The event id must be unique per module and you are free to choose a value. Avoid using predefined event ids (see 2.1.5 *Predefined Events (C/C++)*).

`APMEventId` is declared as:

Type Declaration

```
typedef UInt64 APMEventId;
```

2.2.10 APMEventLevel (C/C++)

`APMEventLevel` is used to specify the importance of an event.

Type Declaration

```
typedef UInt8  APMEventLevel;
```

Possible values are:

- `APM_EVENT_LEVEL_LOG`
- `APM_EVENT_LEVEL_IPC`
- `APM_EVENT_LEVEL_NORMAL`
- `APM_EVENT_LEVEL_DETAIL`
- `APM_EVENT_LEVEL_DEBUG`

2.2.11 APMEventFlags (C/C++)

`APMEventFlags` is used to specify the type of an event.

Type Declaration

```
typedef UInt8  APMEventFlags;
```

Possible values are:

- `APM_FLAG_NONE`
- `APM_FLAG_ENTER`
- `APM_FLAG_LEAVE`
- `APM_FLAG_WAIT`
- `APM_FLAG_WARNING`
- `APM_FLAG_ERROR`

2.2.12 APMParameterType (C/C++)

`APMParameterType` is used to specify the type of the parameter (see 2.5.2 "*Method APMEventArgs (C/C++)*").

Code Definition

```
typedef UInt8  APMParameterType;
```

Possible values are:

- `APM_PARAMETERTYPE_INT32`
- `APM_PARAMETERTYPE_INT64`
- `APM_PARAMETERTYPE_STRING`

2.2.13 EnumAppFilterValuesCallback (C/C++)

Applications may register a callback when calling `APMRegisterApplication` (see 2.3.1 *Method APMRegisterApplication (C/C++)*"). The callback should enumerate all valid filter values by repeatedly calling `APMRegisterFilterValue` (see 2.3.8 "*Method APMRegisterFilterValue (C/C++)*") to register all valid values, that may be selected when starting a Software-Telemetry session. The callback will be triggered by the app.telemetry server, if no filter values are defined for the registered application or once per hour to refresh the values.

Code Definition

```
typedef void (APMFUNC *EnumAppFilterValuesCallback)();
```

2.2.14 APMContext (C/C++)

Values of type `APMContext` need to be passed during inter-process communication to provide the caller context to the invoked program (see 2.4.3 "*Method APMGetContext (C/C++)*" and 2.4.2 "*Method APMAttachContext (C/C++)*").

Code Definition

```
typedef unsigned char APMContext[16];
```

2.2.15 APMSyncMark (C/C++)

Values of type `APMSyncMark` need to be passed during inter-process communication to provide time sequence information (see 2.4.4 "*Method APMGetSyncMark (C/C++)*" and 2.4.5 "*Method APMSetSyncMark (C/C++)*").

Code Definition

```
typedef unsigned char APMSyncMark[16];
```

2.2.16 APMParamHandle (C/C++)

Pointers of type `APMParamHandle` are used to register multiple parameters for use with `APMEventParam` (see 2.5.3 "*Method APMEventParam (C/C++)*" and 2.6 "*Parameter Functions (C/C++)*").

Code Definition

```
typedef void *APMParamHandle;
```

2.3 Registration Functions (C/C++)

2.3.1 Method APMRegisterApplication (C/C++)

Call this function once to provide information about the context of the application.

Syntax

```
APMDECL_EXTERN APMApplicationHandle APMFUNC APMRegisterApplication(
    const char *appName,
    const char *appId,
    const char *appTierName,
    const char *appTierId,
    EnumAppFilterValuesCallback filterCallback);
```

Parameters:

- `appName`: The name of your application (utf-8 encoded)
- `appId`: The Id of your application (utf-8 encoded)
- `appTierName`: The tier inside the application (utf-8 encoded)
- `appTierId`: Id to distinguish multiple services of one application tier (utf-8 encoded)
- `filterCallback`: A callback for filter registration

Return Value:

- This function returns the application handle of the registered application.

Remarks:

Your application must be registered before any events can be fired.

Usage Example

```
void RegisterSoftwareTelemetry()
{
  APMApplicationHandle g_applicationHandle = APMRegisterApplication(
    "app.telemetry Software-Telemetry", "Demo", // appName, appId
    "Backend Services”, "Repository",           // appTierName, -Id
    NULL);                                      // filterCallback
}
```

2.3.2 Method APMRegisterApplicationProperty (C/C++)

Call APMRegisterApplicationProperty to register additional application registration properties immediately after you call to APMRegisterApplication to ensure proper operation.

Syntax

```
APMDECL_EXTERN void APMFUNC APMRegisterApplicationProperty(
    APMApplicationHandle application,
    const char *propertyKey,
    const char *propertyName,
    const char *propertyValue);
```

Parameters:

- `application`: The handle of the application for which you are registering additional registration properties.
- `propertyKey`: The application defined key of the property or the predefined APM_APP_PROPERTY_INSTANCE_GUID_KEY (utf-8 encoded). Limited to 256 Bytes. Keys starting with “apm:” are reserved for internal use.
- `propertyName`: The display name of the property (utf-8 encoded) (APM_APP_PROPERTY_INSTANCE_GUID_NAME for the example above). Limited to 512 Bytes.
- `propertyValue`: The value of the property (utf-8 encoded). Limited to 32768 Bytes.

Return Value:

This function does not return any value.

Remarks:

Application registration properties cannot be changed after they have been set.

Usage Example

```
void RegisterSoftwareTelemetryGUID()
{
  APMApplicationHandle g_applicationHandle = APMRegisterApplication(
    "app.telemetry Software-Telemetry", "Demo", // appName, appId
    "Backend Services”, "Repository",           // appTierName, -Id
    NULL);                                      // filterCallback
  APMRegisterApplicationProperty(g_applicationHandle,
    APM_APP_PROPERTY_INSTANCE_GUID_KEY,
    APM_APP_PROPERTY_INSTANCE_GUID_NAME,
    "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"); // unique for every instance
of this application globally.
```

```
}

void RegisterSoftwareTelemetryCustom()
{
  APMApplicationHandle g_applicationHandle = APMRegisterApplication(
    "app.telemetry Software-Telemetry", "Demo", // appName, appId
    "Backend Services”, "Repository",           // appTierName, -Id
    NULL);                                      // filterCallback
  APMRegisterApplicationProperty(g_applicationHandle,
    "custom:customerId", // used internally and through APIs
    "Customer ID", // displayed as name in the app.telemetry UI
    "132456789");  // displayed as the value in the app.telemtry UI
}
```

2.3.3 Method APMRegisterApplicationValue (C/C++)

Call APMRegisterApplicationValue to register additional static information about your application such as the version number or the display name of this globally unique instance.

Syntax

```
APMDECL_EXTERN void APMFUNC APMRegisterApplicationValue(
    APMApplicationHandle application,
    const char *valueKey,
    const char *valueName,
    const char *value);
```

Parameters:

- `application`: The handle of the application for which you are registering additional registration properties.
- `valueKey`: The application defined key of the value or one of the predefined (APM_APP_VALUE_VERSION_KEY, APM_APP_VALUE_INSTANCE_NAME_KEY, …) (utf-8 encoded). Limited to 256 Bytes. Keys starting with “apm:” are reserved for internal use.
- `valueName`: The display name of the property (utf-8 encoded) (APM_APP_VALUE_VERSION_NAME, APM_APP_VALUE_INSTANCE_NAME_NAME,… for the examples above). Limited to 512 Bytes.
- `value`: The value of the property (utf-8 encoded). Limited to 32768 Bytes.

Return Value:

This function does not return any value.

Remarks:

Application values may be changed at runtime but must not have a high change frequency and do not have any history of previous values.

Usage Example

```
void RegisterSoftwareTelemetry()
{
  APMApplicationHandle g_applicationHandle = APMRegisterApplication(
    "app.telemetry Software-Telemetry", "Demo", // appName, appId
    "Backend Services”, "Repository",           // appTierName, -Id
    NULL);                                      // filterCallback
  APMRegisterApplicationProperty(g_applicationHandle,
```

```
    APM_APP_PROPERTY_INSTANCE_GUID_KEY,
    APM_APP_PROPERTY_INSTANCE_GUID_NAME,
    "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"); // unique for every instance
of this application globally.
  APMRegisterApplicationValue(g_applicationHandle,
    APM_APP_VALUE_VERSION_KEY,
    APM_APP_VALUE_VERSION_NAME,
    "18.3.21");
  APMRegisterApplicationValue(g_applicationHandle,
    APM_APP_VALUE_INSTANCE_NAME_KEY,
    APM_APP_VALUE_INSTANCE_NAME_NAME,
    "Customer X Index Service 1 for E-Mail");
}
```

2.3.4 Method APMUnregisterApplication (C/C++)

Call `APMUnregisterApplication` at the end of your program to tell app.telemetry agent that your program lifetime ends now.

Syntax

```
APMDECL_EXTERN void APMFUNC APMUnregisterApplication(APMApplicationHandle
application);
```

Parameters:

- `application`: The handle of the application which gets unregistered.

Return Value:

This function does not return any value.

Remarks:

Unregistering all applications will stop the connection to the app.telemetry agent. As reconnecting takes time it is not recommended to unregister and reregister applications frequently.

Usage Example

```
void UnregisterSoftwareTelemetry()
{
    APMUnregisterApplication(g_applicationHandle);
}
```

2.3.5 Method APMRegisterModule (C/C++)

Call this function to register modules inside your application.

Syntax

```
APMDECL_EXTERN APMModuleHandle APMFUNC APMRegisterModule(
    const char *modulename);
```

Parameters:

- `modulename` Desired name of the APMModule (utf-8 encoded).

Return Value:

- This function returns a handle which you need to associate events to their corresponding module.

Remarks:

- Different modules are shown as different bars during the analysis.
- Modules are registered per process and not per application, so each application in a process can use modules registered one in that process.
- Modules are identified by their name. A registration of a module with a previously registered module name will return the same module handle as the first module registration as long as the module has not been unregistered.

Usage Example

```
//define constants for your events
#define APM_EVENT_PROCESS_REQUEST  100
#define APM_EVENT_GET_ARTICLE      101
#define APM_EVENT_GET_ARTICLE_DATA 102

APMApplicationHandle g_applicatione;
APMModuleHandle g_module;

void RepositoryRegisterEvents()
{
    APMModuleRegisterEvent(g_module, APM_EVENT_PROCESS_REQUEST, "Process
Request", NULL);
    APMModuleRegisterEvent(g_module, APM_EVENT_GET_ARTICLE, "Get
Article", NULL);
    APMModuleRegisterEvent(g_module, APM_EVENT_GET_ARTICLE_DATA,
"GetArticle::length", "length(bytes)");
}

void RegisterSoftwareTelemetry()
{
    g_module = APMRegisterModule("Repository Handler");
    RepositoryRegisterEvents();
}
```

2.3.6 Method APMUnregisterModule (C/C++)

Call `APMUnregisterModule` if you do not need to trace any more events tied with this module. Calling this function will clean up several internal processes.

Syntax

```
APMDECL_EXTERN void APMFUNC APMUnregisterModule(APMModuleHandle module);
```

Parameters:

- `module`: Handle to the module which should get unregistered

Return Value:

This function does not return any value.

Remarks:

You may register and unregister modules any time but you should not reregister a single module frequently as the app.telemetry server has to keep all module registrations to be able to display all requests of a process correctly.

Usage Example

```
void CleanupModule(APMModuleHandle module)
{
    APMUnregisterModule(module);
}
```

2.3.7 Method APMModuleRegisterEvent (C/C++)

It is required and important to register the events used in your application in order to get the correct event names and parameter descriptions as provided. The `APMModuleRegisterEvent`-function will tell the app.telemetry server a name for every event-ID and optionally parameter descriptions for the parameters used in the event calls.

Syntax

```
APMDECL_EXTERN void APMFUNC APMModuleRegisterEvent(
    APMModuleHandle module,
    APMEventId eventid,
    const char *description,
    const char *parameterdescription);
```

Parameters:

- `module`: Module handle for which you register the event.
- `eventid`: This is the id of the event you want to register.
- `description`: The description of the event.
- `parameterdescription`: Name of event parameters. If more parameters are used in one event, separate the parameter names by semi-colon (`;`). For example: "`name;count;size`"

Return Value:

This function does not return any value.

Remarks:

If an event has more than one parameter you need to separate the `parameterdescription` with semicolons (`;`).

Usage Example

```
void doRegisterEvents()
{
    //given that g_module is a valid handle of a registered module
    APMModuleRegisterEvent(g_module, 121, "Event 1",
"int32;int64;String");
    APMModuleRegisterEvent(g_module, 122, "Event 2", "size;time
stamp;User name");
```

```
}
```

2.3.8 Method APMRegisterFilterValue (C/C++)

It is useful to register all possible filter values used in your application in order to get a list of available filters with descriptions in the client interface.

You may actively register filter values any time after registering the application by calling the `APMRegisterFilterValue` method.

To optimize filter registration in case enumeration of filter values is a costly action, the registration of filter values may be performed in the `EnumAppFilterValuesCallback` function, which is only called if needed (see 2.2.13 "*EnumAppFilterValuesCallback (C/C++)*").

Syntax

```
APMDECL_EXTERN void APMFUNC APMRegisterFilterValue(
    APMApplicationHandle app,
    const char *value,
    const char *description);
```

Parameters:

- `app`: The Handle of the `APMApplication` representing this application.
- `value`: *value* is the raw filter value.
- `description`: This is the description which is presented in the Client-User-Interface as filter.

Return Value:

This function does not return any value.

Remarks:

This function may be called inside the `EnumAppFilterValuesCallback` function which is initiated by the app.telemetry server.

Usage Example

```
void EnumAppFilterValuesCallback()
{
    APMRegisterFilterValue(g_applicationHandle, "raw filter 1",
"Description for filter 1");
}
```

2.3.9 Method APMRegisterCounter (C/C++)

Call `APMRegisterCounter` once for every counter or counter instance to make the counter known to the app.telemetry library for collection.

Syntax

```
APMDECL_EXTERN APMCounterHandle APMFUNC APMRegisterCounter(
  const char *utf8GroupName,
  const char *utf8CounterName,
  APMCounterType counterType,
  APMCounterDataType counterDataType,
  const void *dataAddress,
```

```
  const APMCounterAttribute *counterAttributes);
```

Parameters:

- `utf8GroupName`: The group name of the counter (utf-8 encoded) **Note:** Group names starting with "`apm`" are reserved for internal use and must not be used by application developers.
- `utf8CounterName`: The name of the counter (utf-8 encoded).
- `counterType`: The counter type, see *2.1.9 Counter Types (C/C++)*
- `counterDataType`: The counter data type, see 2.1.10 *Counter Datatypes (C/C++)*.
- `dataAddress`: The address from which the counter value can be read the number of bytes fetched depends on the datatype of the counter.
- `counterAttributes`: List of `APMCounterAttribute` structures with an end marker (`APM_COUNTER_ATTRIBUTE_NONE`) that define additional counter attributes, see 2.1.8 *Counter Attributes (C/C++)* for a list of supported attributes.

Return Value:

- This function returns the handle to the counter that can be used to change some attributes and unregister the counter.

Usage Example

```
static uint32_t g_availableThreadCounter = 0;
void UpdateAvailableThreadCounter(void *data)
{
  g_availableThreadCounter = someCalculation();
}
void RegisterCounters()
{
  const APMCounterCallback callback = { UpdateAvailableThreadCounter,
nullptr, };
  const APMCounterAttribute attributes[] = {
    { APM_COUNTER_ATTRIBUTE_GROUP_DISPLAY_NAME, 0, "Scheduler", },
    { APM_COUNTER_ATTRIBUTE_DISPLAY_NAME, 0, "Threads Available", },
    { APM_COUNTER_ATTRIBUTE_DISPLAY_SUFFIX, 0, "#", },
    { APM_COUNTER_ATTRIBUTE_MEASUREMENT_INTERVAL, 10, nullptr, },
    { APM_COUNTER_ATTRIBUTE_AGGREGATION_FUNCTION_NAME, 0,
APM_COUNTER_AGGREGATION_FUNCTION_MIN, },
    { APM_COUNTER_ATTRIBUTE_CALLBACK, 0, &callback, },
    { APM_COUNTER_ATTRIBUTE_APPLICATION, applicationHandle, nullptr, },
    { APM_COUNTER_ATTRIBUTE_NONE, 0, nullptr, },
  };
  g_numIdleThreadsCounterHandle = APMRegisterCounter("fsc.Scheduler",
"numAvailableThreads", APM_COUNTER_TYPE_RAW, APM_COUNTER_DATATYPE_UINT32,
&g_availableThreadCounter, attributes);
}
```

2.3.10 Method APMUpdateCounterAttributes (C/C++)

Call `APMUpdateCounterAttributes` to update some attributes of a registered counter. **Note:** Only the listed attributes can be updated using this function.

Syntax

```
APMDECL_EXTERN void APMFUNC APMUpdateCounterAttributes(
  APMCounterHandle counterHandle,
```

```
  const APMCounterAttribute *counterAttributes);
```

Parameters:

- `counterHandle`: The handle of a registered counter whose attributes need to be updated.
- `counterAttributes`: List of `APMCounterAttribute` structures with an end marker (`APM_COUNTER_ATTRIBUTE_NONE`) that define attributes to be updated.

Supported Attributes:

- `APM_COUNTER_ATTRIBUTE_GROUP_DISPLAY_NAME`
- `APM_COUNTER_ATTRIBUTE_DISPLAY_NAME`
- `APM_COUNTER_ATTRIBUTE_INSTANCE_DISPLAY_NAME`

Return Value:

This function does not return any value.

Usage Example

```
void UpdateCounters()
{
  const APMCounterAttribute idleThreadsAttributes[] = {
    { APM_COUNTER_ATTRIBUTE_DISPLAY_NAME, 0, "New Display Name", },
    { APM_COUNTER_ATTRIBUTE_NONE, 0, nullptr, },
  };
  APMUpdateCounterAttributes(g_numIdleThreadsCounterHandle,
idleThreadsAttributes);
  const APMCounterAttribute indexedDocumentsAttributes[] = {
    { APM_COUNTER_ATTRIBUTE_INSTANCE_DISPLAY_NAME, 0, "Fileshare
/mnt/xxx", },
    { APM_COUNTER_ATTRIBUTE_NONE, 0, nullptr, },
  };
  APMUpdateCounterAttributes(g_numIndexedDocumentsCounterHandle1,
indexedDocumentsAttributes);
}
```

2.3.11 Method APMUnregisterCounter (C/C++)

Call `APMUnregisterCounter` to unregister a previously registered counter to free it's associated resources and stop any collection.

Syntax

```
APMDECL_EXTERN void APMFUNC APMUnregisterCounter(
  APMCounterHandle counterHandle);
```

Parameters:

- `counterHandle`: The handle of a registered counter that will be unregistered.

Return Value:

- This function does not return any value.

Usage Example

```
void UnregisterCounters()
{
```

```
  APMUnregisterCounter(g_numIdleThreadsCounterHandle;
  APMUnregisterCounter(g_numIndexedDocumentsCounterHandle1);
}
```

2.4 Context Handling Functions (C/C++)

2.4.1 Method APMCreateContext (C/C++)

Call `APMCreateContext` to create a new Software-Telemetry context. If the filter value matches a currently started session filter, logging is started for the current execution thread. Make sure to call `APMReleaseContext` when exiting the context. A context can be regarded as a complete request that is traced through several involved threads or modules. And the full context request has a duration that spans the interval from `CreateContext` to `ReleaseContext`.

Syntax

```
APMDECL_EXTERN void APMFUNC APMCreateContext(APMApplicationHandle app,
const char *filtervalue);
```

Parameters:

- `app`: The Handle of the `APMApplication` representing this application.
- `filtervalue`: (optional) This value is used to match a filter of a started Software-Telemetry session. You should register filter values used by calling `APMRegisterFilterValue` (see 2.3.8 "*Method APMRegisterFilterValue (C/C++)*")

Return Value:

This function does not return any value.

Usage Example

```
void beginRequestProcessing(/* ... */)
{
    APMCreateContext(g_applicationHandle, "filter1");
    //...
    //proceed with application logic and fire events
    //...
    APMReleaseContext();
}
```

2.4.2 Method APMAttachContext (C/C++)

Call `APMAttachContext` to restore the context acquired through `APMGetContext` in another thread or process (see 1.8.1 "*Passing Context*" and 2.4.3 "*Method APMGetContext (C/C++)*").

Syntax

```
APMDECL_EXTERN void APMFUNC APMAttachContext(APMApplicationHandle app,
const APMContext *context);
```

Parameters:

- `app`: The Handle of the `APMApplication` representing this application.

- `context`: The context which you want to attach to.

Return Value:

This function does not return any value.

Usage Example

```
void ContinueRequest(/* ... */, const APMContext *context, APMSyncMark
*syncMark)
{
    APMAttachContext(g_applicationHandle, context);
    //...
    // continue with application logic and fire events
    //...
    APMGetSyncMark(syncMark);
    APMReleaseContext();
}
```

2.4.3 Method APMGetContext (C/C++)

Call the `APMGetContext`-method to get a new context handle.

This context handle is used to synchronize different threads/processes/modules - this helps you track the control flow through a distributed environment. The acquired context handle has to be submitted (either by transmitting over the network or in another way) to the thread/process being called which associates the calling thread with itself by passing the context to the `APMAttachContext`-method.

Syntax

```
APMDECL_EXTERN int APMFUNC APMGetContext(APMContext *context);
```

Parameters:

- `context`: APMGetContext creates a context at the location pointed to by *context*.

Return Value:

- If the function succeeds, the return value is *1* and a context has been saved to the location which `context` pointed to.
- If no context has been created, the return value is *0*.

Usage Example

```
void initiateProcessing(/* ... */)
{
    APMCreateContext(g_applicationHandle, "filterstring");
    APMContext context;
    APMGetContext(&context);

    //pass context to another thread, process on the same or a remote
machine

    doRemote(/* data */, context);

    //do some local processing / wait for the remote processing to
complete
```

```
    APMSyncMark syncMark;
    getRemoteResult(&syncMark);
    APMSetSyncMark(&syncMark);
    APMReleaseContext();
}
```

2.4.4 Method APMGetSyncMark (C/C++)

Call the `APMGetSyncMark`-method to get a synchronization handle that can be passed to another thread/process/module to synchronize control flow.

The difference between a sync mark and a context is:

- A *context* is used to associate with one request. It is created by the initiator and attached to by other threads/process belonging to the same request.
- A *sync mark* is only a time synchronization handle between the different threads/processes which can be used several times to synchronize the control flow exactly (different systems may have different timestamps and different time precision).

Syntax

```
APMDECL_EXTERN int APMFUNC APMGetSyncMark(APMSyncMark *context);
```

Parameters:

- `context`: APMGetSyncMark creates a sync mark at the location pointed to by *context*.

Return Value:

- If the function succeeds, the return value is *1* and a sync mark has been saved to the location which `context` pointed to.
- If no sync-mark has been created, the return value is *0*.

Usage Example

```
void doRemote(/* ... */, APMContext *context)
{
    APMAttachContext(g_applicationHandle, context);

    //application logic, events, ...

    APMSyncMark syncMark;
    APMGetSyncMark(&syncMark);
    sendResultBack(/* ... */, syncMark);
    APMReleaseContext();
}
```

2.4.5 Method APMSetSyncMark (C/C++)

Call `APMSetSyncMark` to pass a sync mark you got from `APMGetSyncMark` for synchronizing the sequence of events (see 1.8.2 “*Synchronizing Timeline with SyncMarks*” and 2.4.4 “*Method APMGetSyncMark (C/C++)*”).

Syntax

```
APMDECL_EXTERN void APMFUNC APMSetSyncMark(const APMSyncMark *context);
```

Parameters:

- `context`: The sync mark which you want to sync with.

Return Value:

This function does not return any value.

Usage Example

```
void initiateProcessing(/* ... */)
{
    APMCreateContext(g_applicationHandle, "filterstring");
    APMContext context;
    APMGetContext(&context);

    //pass context to another thread, process or a remote server

    doRemote(/* data */, context);

    //do some local processing/wait for the remote processing to complete
    APMSyncMark syncMark;
    getRemoteResult(&syncMark);
    APMSetSyncMark(&syncMark);
    APMReleaseContext();
}
```

2.4.6 Method APMReleaseContext (C/C++)

Call `APMReleaseContext` to finish logging and to flush logged information to the app.telemetry infrastructure.
Make sure to call `APMReleaseContext` when exiting the context (when the request is finished).

Syntax

```
APMDECL_EXTERN void APMFUNC APMReleaseContext();
```

Parameters:

This function does not take any arguments.

Return Value:

This function does not return any value.

Remarks:

- If you don't call `APMReleaseContext` the request keeps the *running* status until the request-timeout has been reached.

Usage Example

```
void doRemote(/* ... */, APMContext *context)
{
  APMAttachContext(g_applicationHandle, context);
  //application logic, events, ...
  APMSyncMark syncMark;
  APMGetSyncMark(&syncMark);
  sendResultBack(/* ... */, syncMark);
  APMReleaseContext();
```

```
}
```

2.4.7 Method APMDetachThread (C/C++)

Call the `APMDetachThread`-method when processing of a particular request is suspended and will be resumed later in this or another thread in the same process. Pass the handle returned by this method to the `APMAttachThread`-method to resume processing.

This functionality is in particular useful in queuing systems, where a large number of requests will be queued and processed by a small number of worker threads asynchronous to the initial request. Using the `APMDetach-/APMAttachThread` methods you can follow the request through the process and even measure the time spent in the queue, even if a different thread continues processing.

Syntax

```
APMDECL_EXTERN int APMFUNC APMDetachThread(APMTransactionHandle *handle);
```

Parameters:

- `handle`: `APMDetachThread` generates an id (64bit) that uniquely identifies the current state of the request.

Return Value:

- If the function succeeds, the return value is *1* and a context has been saved to the location which *context* pointed to.
- The function will return *0* in case the library is not available or if no request has been started using `APMCreateContext` and/or `APMAttachContext`.

Remarks:

Be careful to use `APMDetachThread` and `APMAttachThread` in a balanced way. `APMDetachThread` will store the current state of the request in memory and `APMAttachThread` will remove the information from memory while restoring the request state. So calls to `APMDetachThread` lacking a continuation will leak memory and will leave the request in a "running" state. Multiple calls to `APMAttachThread` with the same handle will not have an effect, because the request state is only available to the first call to `APMAttachThread`.

Usage Example

```
struct TaskData
{
    //...
    APMTransactionHandle tx;
    //...
}

void initiateProcessing(/* ... */)
{
    APMCreateContext(g_applicationHandle, "filterstring");
    TaskData *task = new TaskData();
    APMEvent(myModule, APM_EVENT_ENTER_TASK_TO_QUEUE,
APM_EVENT_LEVEL_NORMAL, APM_FLAG_ENTER);
    APMDetachThread(&task->tx);
    queueTask(task);
}

void processTask(TaskData *task)
```

```
{
    APMAttachThread(task->tx);
    APMEvent(myModule, APM_EVENT_REMOVE_TASK_FROM_QUEUE,
APM_EVENT_LEVEL_NORMAL, APM_FLAG_LEAVE);
    // process work
    delete task;
    APMReleaseContext();
}
```

2.4.8 Method APMAttachThread (C/C++)

Call the `APMAttachThread`-method when processing of a particular request is resumed after being suspended in this or another thread of the same process. Pass the handle returned by `APMDetachThread`-method to restore the recorded state of the request.

This functionality is in particular useful in queuing systems, where a large number of requests will be queued and processed by a small number of worker threads asynchronous to the initial request. Using the `Detach-/AttachThread` methods you can follow the request through the process and even measure the time spent in the queue, even if a different thread continues processing.

Syntax

```
APMDECL_EXTERN void APMFUNC APMAttachThread(APMTransactionHandle handle);
```

Parameters:

- `handle`: `APMAttachThread` uses the handle generated by `APMDetachThread` to identify the current state of the request.

Remarks:

See 2.4.7 “*Method APMDetachThread (C/C++)*”

Usage Example

```
struct TaskData
{
    //...
    APMTransactionHandle tx;
    //...
}
void initiateProcessing(/* ... */)
{
    APMCreateContext(g_applicationHandle, "filterstring");
    TaskData *task = new TaskData();
    APMEvent(myModule, APM_EVENT_ENTER_TASK_TO_QUEUE,
APM_EVENT_LEVEL_NORMAL, APM_FLAG_ENTER);
    APMDetachThread(&task->tx);
    queueTask(task);
}
void processTask(TaskData *task)
{
    APMAttachThread(task->tx);
    APMEvent(myModule, APM_EVENT_REMOVE_TASK_FROM_QUEUE,
```

```
APM_EVENT_LEVEL_NORMAL, APM_FLAG_LEAVE);
    // process work
    delete task;
    APMReleaseContext();
}
```

2.5 Event Functions (C/C++)

The four `APMEvent…` functions log Software-Telemetry data. They have the first 4 parameters in common and vary in the options to pass additional parameters.

The common parameters are:

- `module`: The handle to a previously registered module (see 2.3.5 "*Method APMRegisterModule (C/C++)*").
- `eventid`: Id of the event you which you set (see 2.3.7 "*Method APMModuleRegisterEvent (C/C++)*").
- `level`: The log-level for which this event should be logged (see 2.2.10 "*APMEventLevel (C/C++)*").
- `flags`: Use the flags parameter to specify the type of an event (see 2.2.11 "*APMEventFlags (C/C++)*").

2.5.1 Method APMEvent (C/C++)

Call the `APMEvent` function to log an event without parameters. Only the event-ID and the flags define the representation of the event that will be displayed with the registered event name for the used id.

Syntax

```
APMDECL_EXTERN void APMFUNC APMEvent(
    APMModuleHandle module,
    APMEventId eventid,
    APMEventLevel level,
    APMEventFlags flags);
```

Parameters:

- The `APMEvent` function has only the four base parameters (see 2.5 "*Event Functions (C/C++)*").

Return Value:

This function does not return any value.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMEvent(module, id, APM_EVENT_LEVEL_DEBUG, APM_FLAG_NONE);
    //...
```

2.5.2 Method APMEventArgs (C/C++)

Use `APMEventArgs` to log an event with various parameters, which must be passed in a predefined format.

Syntax

```
APMDECL_EXTERN void APMFUNC APMEventArgs(
    APMModuleHandle module,
    APMEventId eventid,
    APMEventLevel level,
    APMEventFlags flags,
    UInt8 paramCount,
    APMParameterType *paramTypes,
    void *paramValues,
    unsigned paramValueSize);
```

Parameters:

- The `APMEventArgs` function has the four base parameters (see 2.5 "*Event Functions (C/C++)*").
- `paramCount`: The number of parameters you want to pass to this function.
- `paramTypes`: This is an array containing the types of the parameters. The size of the array is equal to `paramCount`.
- `paramValues`: This pointer points to the continuous memory region containing the parameters.
- `paramValueSize`: This is the size of the memory region containing the parameter values (`paramValues`).

Return Value:

This function does not return any value.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMParameterType types[3] = {APM_PARAMETERTYPE_INT32,
        APM_PARAMETERTYPE_INT64, APM_PARAMETERTYPE_STRING};

    UInt32 par32 = 4;
    UInt64 par64 = 2147483698004;
    char parstr[] = "genericParam4";
    unsigned int size = (unsigned int)(sizeof(UInt32) + sizeof(UInt64) +
sizeof(UInt16) + strlen(parstr));

    void *mem = calloc(1, size);
    if(mem) {
      char *bptr = (char *)(mem);
      *(UInt32 *)(bptr) = par32;
      bptr += sizeof(UInt32);
      *(UInt64 *)(bptr) = par64;
      bptr += sizeof(UInt64);
      *(UInt16 *)(bptr) = (UInt16)strlen(parstr);
      bptr += sizeof(UInt16);
```

```
      strncpy(bptr, parstr, strlen(parstr));
      APMEventArgs(module, id, APM_EVENT_LEVEL_DEBUG, APM_FLAG_NONE, 3,
types, mem, size);
      free(mem);
    }
    //...
```

2.5.3 Method APMEventParam (C/C++)

Call the `APMEventParam`-method to log events with a generic parameter structure - with any number and combination of textual (string) values and integer values (32- and 64-bit).
To use the `APMEventParam`-method you have to initialize an `APMParamHandle`. With a few helper methods you can add values to this parameter object (see 2.6 "*Parameter Functions (C/C++)*") and use the parameter object in the event function to be logged. Do not forget to free the parameter handle with `APMParamFree` (see 2.6.5 "*Method APMParamFree (C/C++)*").

Syntax

```
APMDECL_EXTERN void APMFUNC APMEventParam(APMModuleHandle module,
    APMEventId eventid,
    APMEventLevel level,
    APMEventFlags flags,
    APMParamHandle param);
```

Parameters:

- The `APMEventParam` function has the four base parameters (see 2.5 "*Event Functions (C/C++)*").
- `param`: This is the handle you obtained from a call to `APMParamInit`.

Return Value:

This function does not return any value.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMParamHandle ph;
    ph = APMParamInit();
    APMParamAdd32(ph, 3);
    APMParamAdd64(ph, 2147483698003);
    APMParamAddString(ph, "genericParam3");
    APMEventParam(module, 123, APM_EVENT_LEVEL_DEBUG, APM_FLAG_NONE, ph);
    APMParamFree(ph);
    //...
```

2.5.4 Method APMEventStr (C/C++)

Use `APMEventStr` to log an event with an additional string value as parameter.

Syntax

```
APMDECL_EXTERN void APMFUNC APMEventStr(
  APMModuleHandle module,
  APMEventId eventid,
  APMEventLevel level,
  APMEventFlags flags,
  const char *param);
```

Parameters:

- The `APMEventStr` function has the four base parameters (see 2.5 "*Event Functions (C/C++)*").
- `param`: Pointer to null-terminated, utf8-encoded string. The maximum string length allowed is 32768 Byte.

Return Value:

This function does not return any value.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMEventStr(module, id, APM_EVENT_LEVEL_DEBUG, APM_FLAG_NONE,
"stringparam1");
    //...
```

2.6 Parameter Functions (C/C++)

The parameter functions provide a way to pass up to 255 parameters to an `APMEventParam` call.

Allocate an `APMParamHandle` by calling `APMParamInit`. Add parameters calling `APMParamAdd32`, `APMParamAdd64` or `APMParamAddString`. Pass the `APMParamHandle` to the `APMEventParam` function and finally call `APMParamFree` to free associated memory.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMParamHandle ph;
    ph = APMParamInit();
    APMParamAdd32(ph, 3);
    APMParamAdd64(ph, 2147483698003);
    APMParamAddString(ph, "genericParam3");
    APMEventParam(module, id, APM_EVENT_LEVEL_DEBUG, APM_FLAG_NONE, ph);
    APMParamFree(ph);
    //...
```

2.6.1 Method APMParamInit (C/C++)

`APMParamInit` is used to initialize an `APMParamHandle`.

Syntax

```
APMDECL_EXTERN APMParamHandle APMFUNC APMParamInit();
```

Parameters:

This function does not take any arguments.

Return Value:

- This function returns a handle which you can use to add parameters and pass to `APMEventParam`.

Remarks:

- Do not forget to call `APMParamFree` (see 2.6.5 "*Method APMParamFree (C/C++)*") to release internal memory.

2.6.2 Method APMParamAdd32 (C/C++)

Call `APMParamAdd32` to add a 32 bit unsigned integer to the parameter handle.

Syntax

```
APMDECL_EXTERN void APMFUNC APMParamAdd32(APMParamHandle param, UInt32
value);
```

Parameters:

- `param`: `APMParamHandle` obtained from `APMParamInit`.
- `value`: The value which you want to add.

Return Value:

This function does not return any value.

2.6.3 Method APMParamAdd64 (C/C++)

Call `APMParamAdd64` to add a 64 bit unsigned integer to the parameter handle.

Syntax

```
APMDECL_EXTERN void APMFUNC APMParamAdd64(APMParamHandle param, UInt64
value);
```

Parameters:

- `param`: `APMParamHandle` obtained from `APMParamInit`.
- `value`: The value which you want to add.

Return Value:

This function does not return any value.

2.6.4 Method APMParamAddString (C/C++)

Call `APMParamAddString` to add a string to the parameter handle.

Syntax

```
APMDECL_EXTERN void APMFUNC APMParamAddString(APMParamHandle param, const
char *value);
```

Parameters:

- `param`: `APMParamHandle` obtained from `APMParamInit`.
- `value`: The value which you want to add. The maximum length of the utf-8 encoded string is 32768 Byte.

Return Value:

This function does not return any value.

2.6.5 Method APMParamFree (C/C++)

Call `APMParamFree` to release the memory allocated for the parameters created by `APMParamInit`.

Syntax

```
APMDECL_EXTERN void APMFUNC APMParamFree(APMParamHandle param);
```

Parameters:

- `param`: `APMParamHandle` created by `APMParamInit`.

Return Value:

This function does not return any value.

Remarks:

Use this function to free internal memory associated to the parameters.

2.7 Fabasoft app.telemetry Button (C/C++)

2.7.1 Method APMReport (C/C++)

Call `APMReport` to create a new feedback report.

The filter value will be matched to the description using the registered filter values. Use the textual description of the feedback entered by the user or generated by the application to identify the report session later again.

If a log pool exists for the current application, the latest requests where the `APMCreateContext` filtervalue parameter matches the filtervalue passed to the `APMReport` function, will be associated with the feedback.

Syntax

```
APMDECL_EXTERN void APMFUNC APMReport(
    APMApplicationHandle app,
    const char *filtervalue,
    const char *reportkey,
    const char *description);
```

Parameters:

- `app`: The Handle of the APMApplication representing this application.
- `filtervalue`: `filtervalue` to match requests with.
- `reportkey`: A key value may be generated by the calling application to associate further descriptions to a feedback. If empty, a new feedback report will be generated on each call and no further information may be attached to the feedback.
- `description`: Textual description of the feedback entered by the user or generated by the application. The maximum length of the description is 32768 Bytes.

Return Value:

This function does not return any value.

Remarks:

- Since version 2009 Fall Release (4.3) the key value is used to associate further descriptions to a feedback.
- In previous versions each call of this function had generated a new feedback report.

Usage Example

```
    //application logic, context created / attached
    APMReport(g_applicationHandle, "filter1", null, "Description entered
by the user / application generated description");
```

2.7.2 Method APMReportValue (C/C++)

Call `APMReportValue` to add a key/value pair to an existing report.

Syntax

```
APMDECL_EXTERN void APMFUNC APMReportValue(
    APMApplicationHandle app,
    const char *reportkey,
    const char *key,
    const char *value);
```

Parameters:

- `app`: The Handle of the APMApplication representing this application.
- `reportkey`: `reportkey` matches the `reportkey` of the `APMReport` function to associate the value with the report session.
- `key`: A key describes the meaning of the value. Only one value can be associated to a single key in one report session.
- `value`: Text (max. 32 kBytes).

Return Value:

This function does not return any value.

Usage Example

```
    //application logic, context created / attached
    APMReport(g_applicationHandle, "filter1", "abc", "Description entered
by the user / application generated description");
    APMReportValue(g_applicationHandle, "abc", "firstname", "John");
```

```
    APMReportValue(g_applicationHandle, "abc", "lastname", "Doe");
```

2.7.3 Method APMReportContent (C/C++)

Call APMReportContent to associate content with an existing report.

Syntax

```
APMDECL_EXTERN void APMFUNC APMReportContent(
    APMApplicationHandle app,
    const char *reportkey,
    const char *filename,
    const void *data,
    UInt64      size);
```

Parameters:

- `app`: The Handle of the APMApplication representing this application.
- `reportkey`: `reportkey` matches the `reportkey` of the `APMReport` function to associate the content with the report session.
- `filename`: A filename describing the content of the file. (max. 256 Bytes)
- `data`: Pointer to a binary data.
- `size`: Size of the `data` content.

Return Value:

- This function does not return any value.

Usage Example

```
    std::string xmlContent = "...";
    //application logic, context created / attached
    APMReport(g_applicationHandle, "filter1", "abc", "Description entered
by the user / application generated description");
    APMReportValue(g_applicationHandle, "abc", "firstname", "John");
    APMReportValue(g_applicationHandle, "abc", "lastname", "Doe");
    APMReportContent(g_applicationHandle, "abc", "parameter.xml",
xmlContent.data(), xmlContent.length())
```

2.7.4 Method APMReportFile (C/C++)

Call APMReportFile to associate a file with an existing report.

Syntax

```
  APMDECL_EXTERN void APMFUNC APMReportFile(
    APMApplicationHandle app,
    const char *reportkey,
    const char *filename,
    FILE *filehandle);
```

Parameters:

- `app`: The handle of the `APMApplication` representing this application.

- `reportkey`: `reportkey` matches the `reportkey` of the `APMReport` function to associate the value with the report session.
- `filename`: A filename describing the content of the file. (max. 256 Bytes)
- `filehandle`: File pointer that has been opened for reading.

Return Value:

This function does not return any value.

Remarks:

- Until version 2010 Spring Release the `filhandle` has been declared as `HANDLE` under Microsoft Windows. This has been changed to a `FILE` pointer for consistency and dependency reasons since version 2010 Summer Release.

Usage Example

```
    //application logic, context created / attached
    APMReport(g_applicationHandle, "filter1", "abc", "Description entered
by the user / application generated description");
    APMReportValue(g_applicationHandle, "abc", "firstname", "John");
    APMReportValue(g_applicationHandle, "abc", "lastname", "Doe");
    FILE *f = fopen("/tmp/parameter.xml", "r");
    APMReportFile(g_applicationHandle, "abc", "parameters.xml", f);
    fclose(f);
```

2.8 Status Functions (C/C++)

2.8.1 Method APMHasActiveContext (C/C++)

Call `APMHasActiveContext` to test if a software-telemetry session is active for some logging level. This information should be used to avoid costly operations preparing parameters for events that are currently not logged.

Syntax

```
APMDECL_EXTERN int APMFUNC APMHasActiveContext(APMEventLevel level);
```

Parameters:

- `level`: The event level of the event you prepare parameters for.

Return Value:

- The return value is non-zero if the event will be logged.

Usage Example

```
void processRequest()
{
    APMCreateContext(g_applicationHandle, "filter value");
    //application logic
    if (0 != APMHasActiveContext(APM_EVENT_LEVEL_DEBUG))
    {
        //prepare parameters, set events,...
    }
```

```
    APMReleaseContext();
}
```

2.8.2 Method APMIsConnected (C/C++)

Call `APMIsConnected` to test, if the application has successfully registered to an app.telemetry agent.

Syntax

```
APMDECL_EXTERN int APMFUNC APMIsConnected();
```

Parameters:

This function does not take any arguments.

Return Value:

- If the application is not connected to an app.telemetry agent this function returns zero.
- If the function is successfully connected, the return value is one.

Usage Example

```
bool checkConnection()
{
    return (1 == APMIsConnected());
}
```

2.8.3 Method APMIsCompatible (C/C++)

Call `APMIsCompatible` to test, if the application was linked with a version that is compatible with the dynamically loaded library.

Syntax

```
APMDECL_EXTERN int APMFUNC APMIsCompatible(UInt32 size, char *message);
```

Parameters:

- `size`: Size in Bytes of the memory region where the second parameter points. Allocate at least **200 Bytes** to get the full version message.
- `message`: Pointer to a memory region of the size specified in the first parameter. A descriptive text with an optional product name and version number will be stored in the allocated memory region.

Return Value:

- This function returns non-zero if the used versions are compatible.

Usage Example

```
bool checkCompatible()
{
  char version[300];
  return (FALSE != APMIsCompatible(300, version));
```

```
}
```

2.9 Explicit Transaction Functions (C/C++)

The functions specified so far all are executed within the context of the current thread. In some cases it is necessary to explicitly define the current transaction context to the context handling functions, for example if your application's processes aren't based on operating system threads.

To use the explicit transaction functions, you have to define the symbolic constant `APM_EXPLICIT_TRANSACTION` in a preprocessor directive for your application. If only `APM_EXPLICIT_TRANSACTION` is defined, only the explicit transaction functions can be used. You have to define `APM_THREAD_TRANSACTION` in addition if you want to use the usual thread context functions from above.

Code Definition

```
#define APM_THREAD_TRANSACTION
#define APM_EXPLICIT_TRANSACTION
```

2.9.1 Method APMTxCreateContext (C/C++)

Call `APMTxCreateContext` to create a new Software-Telemetry context analogously to `APMCreateContext`. You use the `APMTransactionHandle` returned by this function to pass to other explicit transaction functions for the definition of the current transaction context. You have to call `APMTxReleaseContext` with the returned transaction handle when exiting the context.

Syntax

```
APMDECL_EXTERN APMTransactionHandle APMFUNC
APMTxCreateContext(APMTransactionHandle tx, APMApplicationHandle app,
const char *filtervalue);
```

Parameters:

- `tx`: Set this value to 0 in order to create a new transaction handle or use an already existing handle.
- `app`: The Handle of the `APMApplication` representing this application.
- `filtervalue`: (optional) This value is used to match a filter of a started Software-Telemetry session. You should register filter values used by calling `APMRegisterFilterValue` (see 2.3.8 “*Method APMRegisterFilterValue (C/C++)*”)

Return Value:

This function returns the transaction handle for the current or currently created transaction context.

Usage Example

```
void beginRequestProcessing(/* ... */)
{
    APMTransactionHandle tx = APMTxCreateContext(0,g_applicationHandle,
"filter1");
    //...
    //proceed with application logic and fire events
    //...
```

```
    APMTxReleaseContext(tx);
}
```

2.9.2 Method APMTxAttachContext (C/C++)

Call `APMTxAttachContext` to restore the context acquired through `APMGetContext` or `APMTxGetContext` in another thread or process (see 1.8.1 "*Passing Context*" and 2.4.3 "*Method APMGetContext (C/C++)*"). You can attach to an already existing transaction context or create a new transaction context which is returned by `APMTxAttachContext` to pass to other explicit transaction functions.

Syntax

```
APMDECL_EXTERN APMTransactionHandle APMFUNC
APMTxAttachContext(APMTransactionHandle tx, APMApplicationHandle app,
const APMContext *context);
```

Parameters:

- `tx`: Set this value to 0 in order to create a new transaction handle or use an already existing handle.
- `app`: The Handle of the `APMApplication` representing this application.
- `context`: The context which you want to attach to.

Return Value:

This function returns the transaction handle for the current or currently created transaction context.

Usage Example

```
void ContinueRequest(/* ... */, const APMContext *context, APMSyncMark
*syncMark)
{
    APMTransactionHandle tx = APMTxAttachContext(0,g_applicationHandle,
context);
    //...
    // continue with application logic and fire events
    //...
    APMTxGetSyncMark(tx,syncMark);
    APMTxReleaseContext(tx);
}
```

2.9.3 Method APMTxGetContext (C/C++)

Call the `APMTxGetContext`-method to get a new context handle for the current transaction context, analogously to `APMGetContext`.

Syntax

```
APMDECL_EXTERN int APMFUNC APMTxGetContext(APMTransactionHandle tx,
APMContext *context);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.

- `context`: APMGetContext creates a context at the location pointed to by *context*.

Return Value:

- If the function succeeds, the return value is *1* and a context has been saved to the location which `context` pointed to.
- If no context has been created, the return value is *0*.

Usage Example

```
void initiateProcessing(/* ... */)
{
    APMTransactionHandle tx = APMTxCreateContext(0,g_applicationHandle,
"filterstring");
    APMContext context;
    APMTxGetContext(tx,&context);

    //pass context to another thread, process on the same or a remote
machine

    doRemote(/* data */, context);

    //do some local processing / wait for the remote processing to
complete

    APMSyncMark syncMark;
    getRemoteResult(&syncMark);
    APMTxSetSyncMark(tx,&syncMark);
    APMTxReleaseContext(tx);
}
```

2.9.4 Method APMTxGetSyncMark (C/C++)

Call the `APMTxGetSyncMark`-method to get a synchronization handle that can be passed to another thread/process/module to synchronize control flow, same as `APMGetSyncMark`, for the current transaction context.

Syntax

```
APMDECL_EXTERN int APMFUNC APMTxGetSyncMark(APMTransactionHandle tx,
APMSyncMark *context);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.
- `context`: APMGetSyncMark creates a sync mark at the location pointed to by *context*.

Return Value:

- If the function succeeds, the return value is *1* and a sync mark has been saved to the location which `context` pointed to.
- If no sync-mark has been created, the return value is *0*.

Usage Example

```
void doRemote(/* ... */, APMContext *context)
{
    APMTransactionHandle tx = APMTxAttachContext(0,g_applicationHandle,
```

```
context);

    //application logic, events, ...

    APMSyncMark syncMark;
    APMTxGetSyncMark(tx,&syncMark);
    sendResultBack(/* ... */, syncMark);
    APMTxReleaseContext(tx);
}
```

2.9.5 Method APMTxSetSyncMark (C/C++)

Call `APMTxSetSyncMark` to pass a sync mark you got from `APMGetSyncMark` or `APMTxGetSyncMark` for synchronizing the sequence of events (see 1.8.2 "*Synchronizing Timeline with SyncMarks*" and 2.4.4 "*Method APMGetSyncMark (C/C++)*").

Syntax

```
APMDECL_EXTERN void APMFUNC APMTxSetSyncMark(APMTransactionHandle tx,
const APMSyncMark *context);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.
- `context`: The sync mark which you want to sync with.

Return Value:

This function does not return any value.

Usage Example

```
void initiateProcessing(/* ... */)
{
    APMTransactionHandle tx = APMCreateContext(0,g_applicationHandle,
"filterstring");
    APMContext context;
    APMTxGetContext(tx,&context);

    //pass context to another thread, process or a remote server

    doRemote(/* data */, context);

    //do some local processing/wait for the remote processing to complete
    APMSyncMark syncMark;
    getRemoteResult(&syncMark);
    APMTxSetSyncMark(tx,&syncMark);
    APMTxReleaseContext(tx);
}
```

2.9.6 Method APMTxReleaseContext (C/C++)

Call `APMTxReleaseContext` with the current transaction handle to finish logging and exit the context, analogously to `APMReleaseContext`.

Syntax

```
APMDECL_EXTERN void APMFUNC APMTxReleaseContext(APMTransactionHandle tx);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.

Return Value:

This function does not return any value.

Usage Example

```
void doRemote(/* ... */, APMContext *context)
{
  APMTransactionHandle tx = APMTxAttachContext(0, g_applicationHandle,
context);
  //application logic, events, ...
  APMSyncMark syncMark;
  APMTxGetSyncMark(tx, &syncMark);
  sendResultBack(/* ... */, syncMark);
  APMTxReleaseContext(tx);
}
```

2.9.7 Method APMTxHasActiveContext (C/C++)

Call `APMTxHasActiveContext` to test if a software-telemetry session is active for some logging level for the current transaction context, analogously to `APMHasActiveContext`.

Syntax

```
APMDECL_EXTERN int APMFUNC APMTxHasActiveContext(APMTransactionHandle tx,
APMEventLevel level);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.
- `level`: The event level of the event you prepare parameters for.

Return Value:

- The return value is non-zero if the event will be logged.

Usage Example

```
void processRequest()
{
    APMTransactionHandle tx = APMCreateContext(0,g_applicationHandle,
"filter value");
    //application logic
    if (0 != APMTxHasActiveContext(tx,APM_EVENT_LEVEL_DEBUG))
    {
        //prepare parameters, set events,...
    }
    APMTxReleaseContext(tx);
```

```
}
```

2.9.8 Method APMTxEvent (C/C++)

Call the `APMTxEvent` function to log an event without parameters for the current transaction context.

Syntax

```
APMDECL_EXTERN void APMFUNC APMEvent(
    APMTransactionHandle tx,
    APMModuleHandle module,
    APMEventId eventid,
    APMEventLevel level,
    APMEventFlags flags);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.
- The `APMTxEvent` function has the same 4 base parameters as `APMEvent`.

Return Value:

This function does not return any value.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMTxEvent(tx, module, id, APM_EVENT_LEVEL_DEBUG, APM_FLAG_NONE);
    //...
```

2.9.9 Method APMTxEventArgs (C/C++)

Use `APMTxEventArgs` to log an event for the current transaction context with various parameters, which must be passed in a predefined format.

Syntax

```
APMDECL_EXTERN void APMFUNC APMTxEventArgs(
    APMTransactionHandle tx,
    APMModuleHandle module,
    APMEventId eventid,
    APMEventLevel level,
    APMEventFlags flags,
    UInt8 paramCount,
    APMParameterType *paramTypes,
    void *paramValues,
    unsigned paramValueSize);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.

- The `APMTxEventArgs` function has the 4 base parameters (see 2.5 “*Event Functions (C/C++)*”).
- `paramCount`: The number of parameters you want to pass to this function.
- `paramTypes`: This is an array containing the types of the parameters. The size of the array is equal to `paramCount`.
- `paramValues`: This pointer points to the continuous memory region containing the parameters.
- `paramValueSize`: This is the size of the memory region containing the parameter values (`paramValues`).

Return Value:

This function does not return any value.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMParameterType types[3] = {APM_PARAMETERTYPE_INT32,
        APM_PARAMETERTYPE_INT64, APM_PARAMETERTYPE_STRING};

    UInt32 par32 = 4;
    UInt64 par64 = 2147483698004;
    char parstr[] = "genericParam4";
    unsigned int size = (unsigned int)(sizeof(UInt32) + sizeof(UInt64) +
sizeof(UInt16) + strlen(parstr));

    void *mem = calloc(1, size);
    if(mem) {
      char *bptr = (char *)(mem);
      *(UInt32 *)(bptr) = par32;
      bptr += sizeof(UInt32);
      *(UInt64 *)(bptr) = par64;
      bptr += sizeof(UInt64);
      *(UInt16 *)(bptr) = (UInt16)strlen(parstr);
      bptr += sizeof(UInt16);
      strncpy(bptr, parstr, strlen(parstr));

      APMTxEventArgs(tx, module, id, APM_EVENT_LEVEL_DEBUG,
APM_FLAG_NONE, 3, types, mem, size);
      free(mem);
    }
    //...
```

2.9.10 Method APMTxEventParam (C/C++)

Call the `APMTxEventParam`-method to log events with a generic parameter structure analogously to `APMEventParam` for the current transaction context.

Syntax

```
APMDECL_EXTERN void APMFUNC APMTxEventParam(
    APMTransactionHandle tx,
    APMModuleHandle module,
    APMEventId eventid,
```

```
    APMEventLevel level,
    APMEventFlags flags,
    APMParamHandle param);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.
- The `APMTxEventParam` function has the 4 base parameters (see 2.5 "*Event Functions (C/C++)*").
- `param`: This is the handle you obtained from a call to `APMParamInit`.

Return Value:

This function does not return any value.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMParamHandle ph;
    ph = APMParamInit();
    APMParamAdd32(ph, 3);
    APMParamAdd64(ph, 2147483698003);
    APMParamAddString(ph, "genericParam3");
    APMTxEventParam(tx, module, 123, APM_EVENT_LEVEL_DEBUG,
APM_FLAG_NONE, ph);
    APMParamFree(ph);
    //...
```

2.9.11 Method APMTxEventStr (C/C++)

Use `APMTxEventStr` to log an event with an additional string value as parameter.

Syntax

```
APMDECL_EXTERN void APMFUNC APMTxEventStr(
    APMTransactionHandle tx,
    APMModuleHandle module,
    APMEventId eventid,
    APMEventLevel level,
    APMEventFlags flags,
    const char *param);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.
- The `APMTxEventStr` function has the 4 base parameters (see 2.5 "*Event Functions (C/C++)*").
- `param`: Pointer to null-terminated, utf8-encoded string. The maximum string length allowed is 32768 Byte.

Return Value:

This function does not return any value.

Usage Example

```
    //steps required before logging events:
    // * application registered,
    // * module is a handle to a registered module,
    // * id is the id of a registered event,
    // * a context has been created or attached
    APMEventStr(module, id, APM_EVENT_LEVEL_DEBUG, APM_FLAG_NONE,
"stringparam1");
    //...
```

2.9.12 Method APMTxReport (C/C++)

Call `APMTxReport` to create a new feedback report, see 2.7.1 *"Method APMReport (C/C++)"*.

Syntax

```
APMDECL_EXTERN void APMFUNC APMTxReport(
    APMTransactionHandle tx,
    APMApplicationHandle app,
    const char *filtervalue,
    const char *reportkey,
    const char *description);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.
- `app`: The Handle of the APMApplication representing this application.
- `filtervalue`: `filtervalue` to match requests with.
- `reportkey`: A key value may be generated by the calling application to associate further descriptions to a feedback. If empty, a new feedback report will be generated on each call and no further information may be attached to the feedback.
- `description`: Textual description of the feedback entered by the user or generated by the application. The maximum length of the description is 32768 Bytes.

Return Value:

This function does not return any value.

Usage Example

```
    //application logic, context created / attached
    APMTxReport(tx, g_applicationHandle, "filter1", null, "Description
entered by the user / application generated description");
```

2.9.13 Method APMTxReportValue (C/C++)

Call `APMTxReportValue` to add a key/value pair to an existing report.

Syntax

```
APMDECL_EXTERN void APMFUNC APMTxReportValue(
    APMTransactionHandle tx,
    APMApplicationHandle app,
```

```
    const char *reportkey,
    const char *key,
    const char *value);
```

Parameters:

- `tx`: The transaction handle of the current transaction context.
- `app`: The Handle of the APMApplication representing this application.
- `reportkey`: `reportkey` matches the `reportkey` of the `APMReport` function to associate the value with the report session.
- `key`: A key describes the meaning of the value. Only one value can be associated to a single key in one report session.
- `value`: Text (max. 32 kBytes).

Return Value:

This function does not return any value.

Usage Example

```
    //application logic, context created / attached
    APMTxReport(tx, g_applicationHandle, "filter1", "abc", "Description
entered by the user / application generated description");
    APMTxReportValue(tx, g_applicationHandle, "abc", "firstname",
"John");
    APMTxReportValue(tx, g_applicationHandle, "abc", "lastname", "Doe");
```

3 .NET API Reference

The Fabasoft app.telemetry Software-Telemetry .NET API allows you to instrument any .NET application with instrumentation points to see what's going on in complex applications.

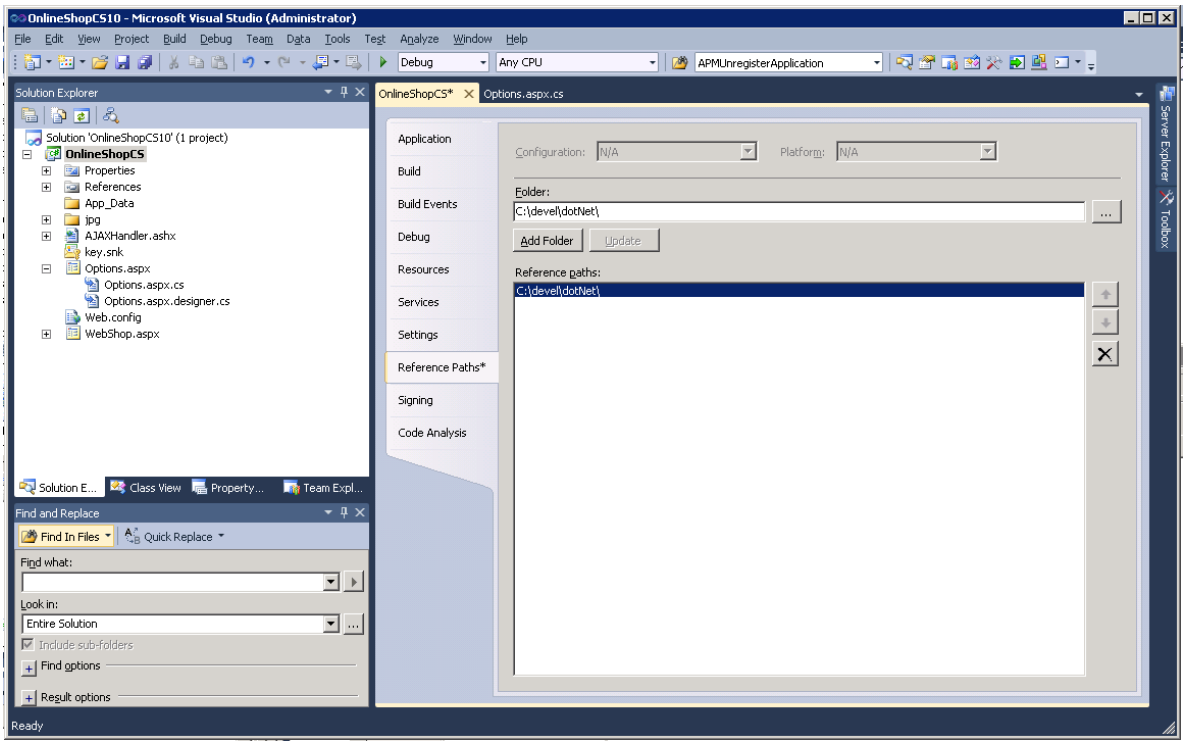
Since version 2023 the Software-Telemetry API supports .NET 6 whereas .NET Framework is no longer supported.

3.1 Configure Visual Studio .NET Project

In order to use the Software-Telemetry API in your Microsoft Visual Studio .NET project you have to include app.telemetry API .NET library `softwaretelemetry.dll` which comes with the app.telemetry API developer resources.

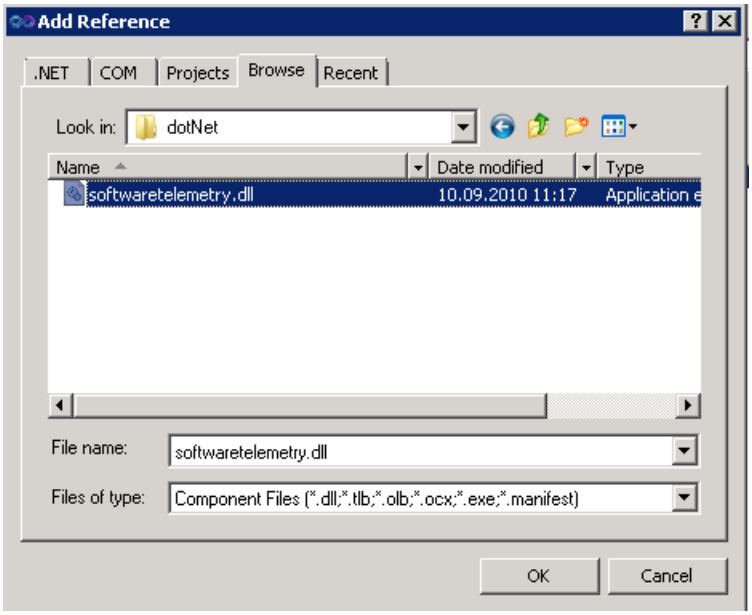
Follow the described steps to include the required resources.

In the Solution Explorer you have to open the context-menu on the project which should be analyzed and click "*Properties*". Choose the tab "*Reference Paths*". Browse to the Folder where "`softwaretelemetry.dll`" is located (which is delivered with the app.telemetry Software-Telemetry SDK resources) and add this folder to the Reference paths list.



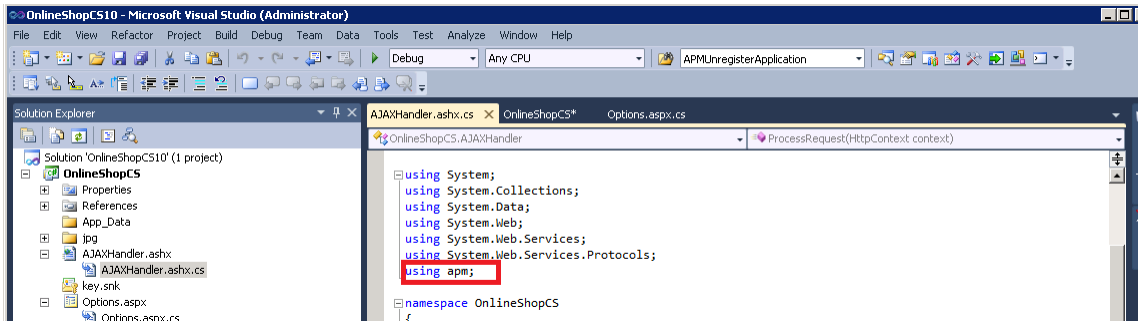
You also have to add the `softwaretelemetry.dll` library to the project references.

Select the *References*-folder in the Solution Explorer of your project and open the context menu and click “*Add Reference ...*“. Select the `softwaretelemetry.dll` library via the "*Browse*" tab from the same location as configured above.

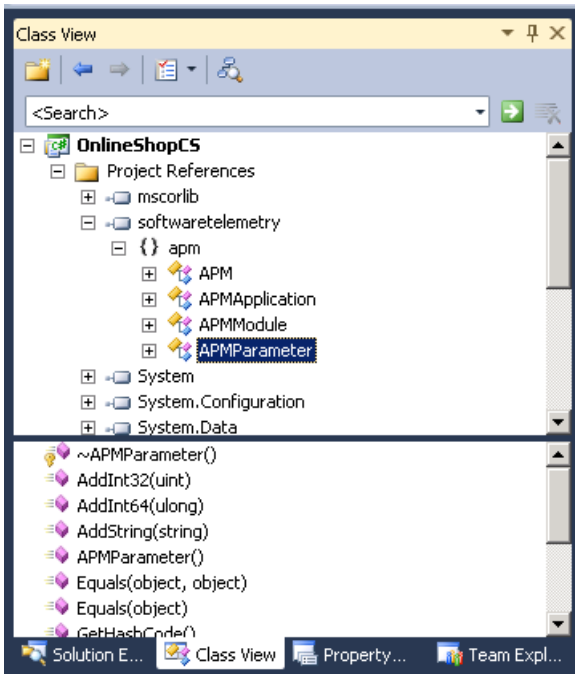


After the references to the `softwaretelemetry.dll` library is configured correctly you can use the library like this:

```
using apm;
```



Here you can see an overview of the app.telemetry Software-Telemetry SDK APM classes inside Microsoft Visual Studio:



The .NET Software-Telemetry SDK is defined in the namespace "`apm`" which has to be imported by the statement "`using apm;`" in order to use any constants or methods from the classes defined in the SDK.

3.1.1 NuGet Package

Since version 2023 the .NET Software-Telemetry SDK is published additionally as "`softwaretelemetry`" NuGet Package.

3.2 Constants (.NET)

All the constants are static members of the *APM* class and can be accessed in a static way.

3.2.1 Flags (.NET)

Flags are used to specify the type of an event and are passed as argument to every event call. They are predefined by the API on the base class `APM` with the data type `Byte` and have the following possible values:

- `FLAG_NONE`
- `FLAG_ENTER`
- `FLAG_LEAVE`
- `FLAG_WAIT`
- `FLAG_WARNING`
- `FLAG_ERROR`
- `FLAG_PROCESS_INFO`

For more details about the flags and there meaning see the general chapter 1.7.1 "*Flags*".

3.2.2 Log Level (.NET)

An event will only be logged if the event-level is lower or equal to the value selected in the session or log-definition. The log levels are predefined by the API on the base class `APM` with the data type `Byte` and have the following possible values:

- `EVENT_LEVEL_LOG`
- `EVENT_LEVEL_IPC` (only internal)
- `EVENT_LEVEL_NORMAL`
- `EVENT_LEVEL_DETAIL`
- `EVENT_LEVEL_DEBUG`

For more details about the log levels and there meaning see the general chapter 1.7.2 “*Log Level*”.

3.2.3 Predefined Modules (.NET)

To use a predefined module create a new `APMModule` object (by means of calling the constructor) with the predefined module name. This object will be used later to fire events for that module.

For a list of available standard modules with a set of predefined events see chapter 1.5 “*Standard Modules*”.

Usage Example

```
APMModule mod1 = new APMModule(APM.MODULE_NAME_OS);
APMModule mod2 = new APMModule(APM.MODULE_NAME_XMLHTTP);
```

3.2.4 Predefined Events (.NET)

The SDK already provides a set of predefined events used for general purpose. For a full listing of all available predefined events see the generic chapter 1.6 "*Predefined Events*".

All predefined events are declared and available as global constants on the base class “`APM`”.

Error/Trace Events:

Use these standard events to mark error, warning or info conditions in your application:

- `EVENT_ERROR`
- `EVENT_WARNING`
- `EVENT_INFO`
- `EVENT_TRACE`

3.2.5 Predefined Application Property Keys and Names (.NET)

View the language agnostic Application Properties section for an explanation of the properties listed below.

All predefined application property keys and names are available as public constants on the APM class.

- `PROPERTY_INSTANCE_GUID_KEY` with name `PROPERTY_INSTANCE_GUID_NAME`

3.2.6 Predefined Application Value Keys and Names (.NET)

View the language agnostic Application Values section for an explanation of the properties listed below.

All predefined application value keys and names are available as public constants on the APM class.

- `VALUE_VERSION_KEY` with name `VALUE_VERSION_NAME`
- `VALUE_INSTANCE_NAME_KEY` with name `VALUE_INSTANCE_NAME_NAME`

3.2.7 Aggregation Function Names (.NET)

The Aggregation Function Names are static members of the *APMCounterOptions* class and can be accessed in a static way.

Name	Description
`COUNTER_AGGREGATION_FUNCTION_SUM`	Sum
`COUNTER_AGGREGATION_FUNCTION_AVG`	Average
`COUNTER_AGGREGATION_FUNCTION_MAX`	Maximum
`COUNTER_AGGREGATION_FUNCTION_MIN`	Minimum

3.3 Data Types / Class Overview (.NET)

3.3.1 APM (.NET)

The `APM` class is the base class of the Software-Telemetry SDK. It contains all global defined constants as described in chapter 0„*The .NET Software-Telemetry* SDK is defined in the namespace "`apm`" which has to be imported by the statement "`using apm;`" in order to use any constants or methods from the classes defined in the SDK.

3.3.2 NuGet Package

Since version 2023 the .NET Software-Telemetry SDK is published additionally as "`softwaretelemetry`" NuGet Package.

Constants (.NET)". Use the methods from the `APM` class in a static way (there is no new instance allocation required).

The `APM` class supports the following global static methods which do not require an application scope or status:

- `Connect`
- `Disconnect`
- `IsConnected`
- `IsCompatible`
- `DetachThread`
- `AttachThread`
- `RegisterCounter`
- `UpdateCounter`

- `UnregisterCounter`

A detailed description of this class with all methods explained is described in chapter 3.3.11 "*ComputeCounter Delegates (.NET)*

The counter value is provided to the app.telemetry library as return value of one of the following delegates:

Syntax

```
public delegate UInt32 ComputeCounterUInt32();
public delegate UInt64 ComputeCounterUInt64();
public delegate String ComputeCounterString();
```

These delegates are used as parameters in the method `RegisterCounter` of the `APM` class.

3.3.3 APMCounterOptions (.NET)

The `APMCounterOptions` class contains counter attributes as fields. Counter attributes define behavior, aggregation, measurement, representation and labels of registered counters. Some of the attributes can be changed but most must be defined during the initial counter registration in the method `RegisterCounter` of the `APM` class.

Name	Description	Type
`GroupDisplayName`	Group display name of the counter.	string
`DisplayName`	Display name of the counter.	string
`InstanceName`	Name of the counter instance.	string
`InstanceDisplayName`	Instance display name of the counter.	string
`DisplaySuffix`	Value suffix.	string
`AggregationFunctionName`	Aggregation function, see 3.2.7 *"Aggregation Function Names (.NET)"*	string
`WarningLimit`	Warning limit for service check, related to the service check property "warning".	string
`ErrorLimit`	Critical limit for service check, related to the service check property "error".	string
`FractionBase`	Base for fractional counter values.	UInt64
`Application`	`ApplicationHandle` of the application that owns the counter.	UInt32
`MeasurementInterval`	Interval (in seconds) of automatic counter recording.	uint

3.3.4 APMUpdateCounterOptions (.NET)

The `APMUpdateCounterOptions` class is similar to the `APMCounterOptions` class, but it contains only the counter attributes that may be updated for a registered counter.

Name	Description	Type
`GroupDisplayName`	Group display name of the counter.	string
`DisplayName`	Display name of the counter.	string
`InstanceDisplayName`	Instance display name of the counter.	string

APM Class and Methods (.NET)".

3.3.5 APMApplication (.NET)

`APMApplication` is the base class for own application implementation and handles the request scope and reporting functions.

After creating an instance with the class constructor passing the application registration information, it stores the application handle inside and provides all context-specific functions for your application.

The `APMApplication` class supports the following instance/class methods which require an application scope and have to be called on your object instance of `APMApplication`:

- `RegisterFilterValue`
- `RegisterApplicationProperty`
- `RegisterApplicationValue`
- `CreateContext`
- `AttachContext`
- `ReleaseContext`
- `GetContext`
- `GetSyncMark`
- `SetSyncMark`
- `HasActiveContext`
- `Report`
- `ReportValue`
- `ReportContent`
- `ReportFile`

A detailed description of this class with all methods explained is described in chapter 3.4.10 "*Method RegisterCounter (.NET)*

Call RegisterCounter once for every counter or counter instance to make the counter known to the app.telemetry library for collection. There exist three overloaded methods of RegisterCounter corresponding to the three possible data types for counter, UInt32, UInt64 and String.

Syntax

```
public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterUInt32 fn, APMCounterOptions? options
= null)
```

```
public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterUInt64 fn, APMCounterOptions? options
= null)

public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterString fn, APMCounterOptions? options
= null)
```

Parameters:

- `group`: The group name of the counter. Note: Group names starting with “apm” are reserved for internal use and must not be used by application developers.
- `name`: The name of the counter.
- `type`: The counter type, see 3.3.10 *"APMCounterType (.NET)"*.
- `fn`: The ComputeCounter delegate, see 3.3.11 *"ComputeCounter Delegates (.NET)"*.
- `options`: Instance of the `APMCounterOptions` class that defines additional counter attributes, see 3.3.12 *"APMCounterOptions (.NET)"*.

Return Value:

This function returns the handle to the counter that can be used to change some attributes and to unregister the counter again.

Usage Example

```
class Counter
{
  static Random rnd = new();
  public static UInt32 ComputeUInt32Counter() => (UInt32)rnd.Next(10);
  internal Uint32 counterHandle = 0;

  public void RegisterCounter()

  {
    APMCounterOptions options = new()
    {
      GroupDisplayName = ".NET Counter",
      MeasurementInterval = 10
    };
    counterHandle = APM.RegisterCounter("dotnetGroup", "dotnetCounter",
APMCounterType.APMCounterTypeRaw, ComputeUInt32Counter, options));
  }
}
```

3.3.6 Method UpdateCounter (.NET)

Call `UpdateCounter` to update some of the attributes of a registered counter. The attributes that can be updated are defined as fields of the `APMUpdateCounterOptions` (see 3.3.13 *"APMUpdateCounterOptions (.NET)"*) class:

- `GroupDisplayName`
- `DisplayName`
- `InstanceDisplayName`

Syntax

```
public static void UpdateCounter(Uint32 counterHandle,
```

```
APMUpdateCounterOptions options)
```

Parameters:

- `counterHandle`: The handle of the registered counter whose attributes need to be updated, returned by `RegisterCounter`
- `options`: Instance of the `APMUpdateCounterOptions` class that defines the counter attributes to be updated.

Return Value:

This function does not return any value.

Usage Example

```
class Counter
{
  // …

  public void UpdateCounter()
  {
    APMUpdateCounterOptions options = new()
    {
      GroupDisplayName = "Nice Counter Display Name",
    };
    APM.UpdateCounter(counterHandle, options));
  }
}
```

3.3.7 Method UnregisterCounter (.NET)

Call `UnregisterCounter` to unregister a previously registered counter to free it's associated resources and stop any collection.

Syntax

```
public static void UnregisterCounter(Uint32 counterHandle)
```

Parameters:

- `counterHandle`: The handle of the registered counter that will be unregistered.

Return Value:

This function does not return any value.

Usage Example

```
class Counter
{
  // …

  public void UnregisterCounter()
  {
    APM.UnregisterCounter(counterHandle));
```

```
    }
}
```

APMApplication Class and Methods (.NET)".

3.3.8 APMModule (.NET)

`APMModule` is the base class for own module implementation and is used to fire instrumentation events.

After creating an instance with the class constructor passing the module name, it stores the module handle inside and provides all module-specific functions for your application.

The `APMModule` class supports the following instance/class methods which require a module scope and have to be called on your object instance of `APMModule`:

- `RegisterEvent`
- `Event`
- `EventStr`
- `EventParam`

A detailed description of this class with all methods explained is described in chapter 3.6 "*APMModule Class and Methods (.NET)*".

3.3.9 APMParameter (.NET)

`APMParameter` is used as data class for generic event parameters.

To define any parameters used in events you have to create a new `APMParameter` instance and add the typed parameters to that instance:

- `AddInt32`
- `AddInt64`
- `AddString`

A detailed description of this class with all methods explained is described in chapter 3.7 "*APMParameter Class and Methods (.NET)*".

3.3.10 APMCounterType (.NET)

The `enum APMCounterType` specifies the type of counter registered using the method `RegisterCounter` of the `APM` class.

Name	Description
`APMCounterTypeRaw`	Raw counter without special interpretation.
`APMCounterTypeCounter`	Per second values.
`APMCounterTypeFraction`	Percent value.

3.3.11 ComputeCounter Delegates (.NET)

The counter value is provided to the app.telemetry library as return value of one of the following delegates:

Syntax

```
public delegate UInt32 ComputeCounterUInt32();
public delegate UInt64 ComputeCounterUInt64();
public delegate String ComputeCounterString();
```

These delegates are used as parameters in the method `RegisterCounter` of the `APM` class.

3.3.12 APMCounterOptions (.NET)

The `APMCounterOptions` class contains counter attributes as fields. Counter attributes define behavior, aggregation, measurement, representation and labels of registered counters. Some of the attributes can be changed but most must be defined during the initial counter registration in the method `RegisterCounter` of the `APM` class.

Name	Description	Type
`GroupDisplayName`	Group display name of the counter.	string
`DisplayName`	Display name of the counter.	string
`InstanceName`	Name of the counter instance.	string
`InstanceDisplayName`	Instance display name of the counter.	string
`DisplaySuffix`	Value suffix.	string
`AggregationFunctionName`	Aggregation function, see 3.2.7 *"Aggregation Function Names (.NET)"*	string
`WarningLimit`	Warning limit for service check, related to the service check property “warning”.	string
`ErrorLimit`	Critical limit for service check, related to the service check property “error”.	string
`FractionBase`	Base for fractional counter values.	UInt64
`Application`	`ApplicationHandle` of the application that owns the counter.	UInt32
`MeasurementInterval`	Interval (in seconds) of automatic counter recording.	uint

3.3.13 APMUpdateCounterOptions (.NET)

The `APMUpdateCounterOptions` class is similar to the `APMCounterOptions` class, but it contains only the counter attributes that may be updated for a registered counter.

Name	Description	Type
`GroupDisplayName`	Group display name of the counter.	string

`DisplayName`	Display name of the counter.	string
`InstanceDisplayName`	Instance display name of the counter.	string

3.4 APM Class and Methods (.NET)

The `APM` class is the base class of the Software-Telemetry SDK. It contains all global defined constants as described in chapter 0„*The .NET Software-Telemetry* SDK is defined in the namespace "`apm`" which has to be imported by the statement "`using apm;`" in order to use any constants or methods from the classes defined in the SDK.

3.4.1 NuGet Package

Since version 2023 the .NET Software-Telemetry SDK is published additionally as "`softwaretelemetry`" NuGet Package.

Constants (.NET)". Use the methods from the `APM` class in a static way (there is no new instance allocation required).

Remarks:

With version 2010 Fall Release the application-specific functions (`CreateContext`, ..., `Report`, ...) of the .Net API where moved to the `APMApplication` class (see chapter 3.4.10 "*Method RegisterCounter (.NET)*

Call RegisterCounter once for every counter or counter instance to make the counter known to the app.telemetry library for collection. There exist three overloaded methods of RegisterCounter corresponding to the three possible data types for counter, UInt32, UInt64 and String.

Syntax

```
public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterUInt32 fn, APMCounterOptions? options
= null)

public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterUInt64 fn, APMCounterOptions? options
= null)

public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterString fn, APMCounterOptions? options
= null)
```

Parameters:

- `group`: The group name of the counter. Note: Group names starting with "apm" are reserved for internal use and must not be used by application developers.
- `name`: The name of the counter.
- `type`: The counter type, see 3.3.10 *"APMCounterType (.NET)"*.
- `fn`: The ComputeCounter delegate, see 3.3.11 *"ComputeCounter Delegates (.NET)"*.
- `options`: Instance of the `APMCounterOptions` class that defines additional counter attributes, see 3.3.12 *"APMCounterOptions (.NET)"*.

Return Value:

This function returns the handle to the counter that can be used to change some attributes and to unregister the counter again.

Usage Example

```
class Counter
{
  static Random rnd = new();
  public static UInt32 ComputeUInt32Counter() => (UInt32)rnd.Next(10);
  internal Uint32 counterHandle = 0;

  public void RegisterCounter()

  {
    APMCounterOptions options = new()
    {
      GroupDisplayName = ".NET Counter",
      MeasurementInterval = 10
    };
    counterHandle = APM.RegisterCounter("dotnetGroup", "dotnetCounter",
APMCounterType.APMCounterTypeRaw, ComputeUInt32Counter, options));
  }
}
```

3.4.2 Method UpdateCounter (.NET)

Call `UpdateCounter` to update some of the attributes of a registered counter. The attributes that can be updated are defined as fields of the `APMUpdateCounterOptions` (see 3.3.13 *"APMUpdateCounterOptions (.NET)"*) class:

- `GroupDisplayName`
- `DisplayName`
- `InstanceDisplayName`

Syntax

```
public static void UpdateCounter(Uint32 counterHandle,
APMUpdateCounterOptions options)
```

Parameters:

- `counterHandle`: The handle of the registered counter whose attributes need to be updated, returned by `RegisterCounter`
- `options`: Instance of the `APMUpdateCounterOptions` class that defines the counter attributes to be updated.

Return Value:

This function does not return any value.

Usage Example

```
class Counter
{
  // …


  public void UpdateCounter()

  {
    APMUpdateCounterOptions options = new()
```

```
    {
      GroupDisplayName = "Nice Counter Display Name",
    };
    APM.UpdateCounter(counterHandle, options));
  }
}
```

3.4.3 Method UnregisterCounter (.NET)

Call `UnregisterCounter` to unregister a previously registered counter to free it's associated resources and stop any collection.

Syntax

```
public static void UnregisterCounter(Uint32 counterHandle)
```

Parameters:

- `counterHandle`: The handle of the registered counter that will be unregistered.

Return Value:

This function does not return any value.

Usage Example

```
class Counter
{
  // …

  public void UnregisterCounter()
  {
    APM.UnregisterCounter(counterHandle));
  }
}
```

APMApplication Class and Methods (.NET)") and have to be called on an object instance of this class!

The Methods `Connect/Disconnect` and all counter functionality (`RegisterCounter/UpdateCounter/UnregisterCounter`) is available since version 2023.

3.4.4 Method Connect (.NET)

Call `Connect` to establish a connection to the app.telemetry agent.

Syntax

```
public static void Connect();
```

Parameters:

This function does not take any arguments.

Return Value:

This function does not return any value.

3.4.5 Method Disconnect (.NET)

Call `Disconnect` to disconnect from the app.telemetry agent, this will also cleanup the app.telemetry communication thread and will disable automatic reconnection. To reestablish the connection to the app.telemetry agent call `Connect`.

Syntax

```
public static void Disconnect();
```

Parameters:

This function does not take any arguments.

Return Value:

This function does not return any value.

3.4.6 Method IsConnected (.NET)

The API method `IsConnected` will tell you if the application (the one that is instrumented with the APM SDK) has successfully registered to an app.telemetry agent and is currently connected to the agent process.

Only when an application is successfully connected to an app.telemetry agent, Software-Telemetry data will be processed and sent to and collected by the app.telemetry server

Syntax

```
public static Boolean IsConnected();
```

Parameters:

This function does not take any arguments.

Return Value:

- `true`: if the application has successfully registered to and connected to an app.telemetry agent.
- `false`: if the application has not registered or is currently not connected to an app.telemetry agent or if the native Software-Telemetry library is not loaded correctly.

Usage Example

```
private void isConnected_Click(object sender, EventArgs e)
{
    isConnectedResult.Text = (APM.IsConnected()) ? "yes" : "no";
}
```

3.4.7 Method IsCompatible (.NET)

Call `IsCompatible` to test, if the application was linked with a version that is compatible with the dynamically loaded library.

Syntax

```
public static Boolean IsCompatible(ref String message);
```

Parameters:

- `message (String)`: An output parameter containing the result of the `IsCompatible` check.

Return Value:

- `true`: if the used versions are compatible
- `false` will be returned otherwise (versions not compatible) – the message object will contain details about the incompatibility

Usage Example

```
String message = "";
if (!APM.IsCompatible(ref message)) {
    //incompatible
    Console.WriteLine(message);
}
```

3.4.8 Method DetachThread (.NET)

Call the `DetachThread`-method when processing of a particular request is suspended and will be resumed later in this or another thread in the same process. Pass the handle returned by this method to the `AttachThread`-method to resume processing (see 3.4.9 "*Method AttachThread (.NET)*").

This functionality is in particular useful in queuing systems, where a large number of requests will be queued and processed by a small number of worker threads asynchronous to the initial request. Using the *Detach-/AttachThread* methods you can follow the request through the process and even measure the time spent in the queue, even if a different thread continues processing.

Syntax

```
public static Boolean DetachThread(out UInt64 transaction)
```

Parameters:

- `transaction`: `DetachThread` generates an id (64bit) that uniquely identifies the current state of the request.

Return Value:

- If the function succeeds, the return value is `true` and the transaction handle has been filled.
- The function will return `false` in case the library is not available or if no request has been started using `CreateContext` and/or `AttachContext`.

Usage Example

```
public class Task {
    //...
    public UInt64 tx;
    //...
    public void Initialize(/* ... */)
    {
        app.CreateContext("filterstring");
        module.Event(EVENT_ENTER_TASK_TO_QUEUE, APM.EVENT_LEVEL_NORMAL,
```

```
APM.FLAG_ENTER);
        APM.DetachThread(out tx);
        schedule();

    }

    public void process()
    {

        APM.AttachThread(tx);
        module.Event(EVENT_REMOVE_TASK_FROM_QUEUE,
APM.EVENT_LEVEL_NORMAL, APM.FLAG_LEAVE);

        // process work

        app.ReleaseContext();
    }
}
```

Remarks:

Be careful to use `DetachThread` and `AttachThread` in a balanced way. `DetachThread` will store the current state of the request in memory and `AttachThread` will remove the information from memory while restoring the request state. So calls to `DetachThread` lacking a continuation will leak memory and will leave the request in a "*running*" state. Multiple calls to `AttachThread` will not have an effect, because the request state is only available to the first call to `AttachThread`.

3.4.9 Method AttachThread (.NET)

Call the `AttachThread`-method when processing of a particular request is resumed after being suspended in this or another thread of the same process. Pass the handle returned by `DetachThread`-method to restore the recorded state of the request.

For more details see 3.4.8 “*Method DetachThread (.NET)*”.

Syntax

```
public static void AttachThread(UInt64 transaction);
```

Parameters:

- `transaction`: `AttachThread` uses the transaction handle generated by `DetachThread` to identify the current state of the request.

Return Value:

This function does not return any value.

Usage Example

`See` 3.4.8 “*Method DetachThread (.NET)*”

3.4.10 Method RegisterCounter (.NET)

Call `RegisterCounter` once for every counter or counter instance to make the counter known to the app.telemetry library for collection. There exist three overloaded methods of RegisterCounter corresponding to the three possible data types for counter, UInt32, UInt64 and String.

Syntax

```
public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterUInt32 fn, APMCounterOptions? options
= null)

public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterUInt64 fn, APMCounterOptions? options
= null)

public static UInt32 RegisterCounter(String group, String name,
APMCounterType type, ComputeCounterString fn, APMCounterOptions? options
= null)
```

Parameters:

- `group`: The group name of the counter. Note: Group names starting with “apm” are reserved for internal use and must not be used by application developers.
- `name`: The name of the counter.
- `type`: The counter type, see 3.3.10 *"APMCounterType (.NET)"*.
- `fn`: The ComputeCounter delegate, see 3.3.11 *"ComputeCounter Delegates (.NET)"*.
- `options`: Instance of the `APMCounterOptions` class that defines additional counter attributes, see 3.3.12 *"APMCounterOptions (.NET)"*.

Return Value:

This function returns the handle to the counter that can be used to change some attributes and to unregister the counter again.

Usage Example

```
class Counter
{
  static Random rnd = new();
  public static UInt32 ComputeUInt32Counter() => (UInt32)rnd.Next(10);
  internal Uint32 counterHandle = 0;

  public void RegisterCounter()

  {
    APMCounterOptions options = new()
    {
      GroupDisplayName = ".NET Counter",
      MeasurementInterval = 10
    };
    counterHandle = APM.RegisterCounter("dotnetGroup", "dotnetCounter",
APMCounterType.APMCounterTypeRaw, ComputeUInt32Counter, options));
  }
}
```

3.4.11 Method UpdateCounter (.NET)

Call `UpdateCounter` to update some of the attributes of a registered counter. The attributes that can be updated are defined as fields of the `APMUpdateCounterOptions` (see 3.3.13 *"APMUpdateCounterOptions (.NET)"*) class:

- `GroupDisplayName`
- `DisplayName`

- `InstanceDisplayName`

Syntax

```
public static void UpdateCounter(Uint32 counterHandle,
APMUpdateCounterOptions options)
```

Parameters:

- `counterHandle`: The handle of the registered counter whose attributes need to be updated, returned by `RegisterCounter`
- `options`: Instance of the `APMUpdateCounterOptions` class that defines the counter attributes to be updated.

Return Value:

This function does not return any value.

Usage Example

```
class Counter
{
  // …

  public void UpdateCounter()
  {
    APMUpdateCounterOptions options = new()
    {
      GroupDisplayName = "Nice Counter Display Name",
    };
    APM.UpdateCounter(counterHandle, options));
  }
}
```

3.4.12 Method UnregisterCounter (.NET)

Call `UnregisterCounter` to unregister a previously registered counter to free it's associated resources and stop any collection.

Syntax

```
public static void UnregisterCounter(Uint32 counterHandle)
```

Parameters:

- `counterHandle`: The handle of the registered counter that will be unregistered.

Return Value:

This function does not return any value.

Usage Example

```
class Counter
{
  // …
```

```
  public void UnregisterCounter()
  {
    APM.UnregisterCounter(counterHandle));
  }
}
```

3.5 APMApplication Class and Methods (.NET)

The `APMApplication` class is the base class for your application registration, the context- and report-functions and the filter registration (callback or directly). It stores the application handle inside and provides all application-specific functions for your application to be called on your `APMApplication` object instance.

Your own application implementation may override the callback function `EnumFilterValuesCallback` (default implementation is empty) to provide the filters values via the callback function. Alternatively the filters may also be registered directly.

The application is registered when a new instance of the APMApplication class is being created.

3.5.1 Constructor APMApplication (.NET)

Syntax

```
public APMApplication(string appName, string appId,
                      string appTier, string appTierId);
```

Parameters:

- `appName`: The name of your application
- `appId`: The id of your application
- `appTier`: The tier inside the application
- `appTierId`: id to distinguish multiple services of one application tier

3.5.2 Property ApplicationHandle (.NET)

`ApplicationHandle` lets you obtain the corresponding `UInt32` handle of the application, it is read-only.

3.5.3 Method RegisterApplicationProperty (.NET)

Call RegisterApplicationProperty to register additional application registration properties immediately after you call to the APMApplication class constructor to ensure proper operation.

Syntax

```
public void RegisterApplicationProperty(string propertyKey, string
propertyName, string propertyValue);
```

Parameters:

- `propertyKey`: The application defined key of the property or the predefined PROPERTY_INSTANCE_GUID_KEY. Limited to 256 utf-8 encoded Bytes. Keys starting with “apm:” are reserved for internal use.

- `propertyName`: The display name of the property (PROPERTY_INSTANCE_GUID_NAME for the example above). Limited to 512 utf-8 encoded Bytes.
- `propertyValue`: The value of the property. Limited to 32768 utf-8 encoded Bytes.

Remarks:

Application registration Properties cannot be changed after they have been set.

3.5.4 Method RegisterApplicationValue (.NET)

Syntax

```
public void RegisterApplicationValue(string valueKey, string valueName,
string value);
```

Parameters:

- `valueKey`: The application defined key of the value or one of the predefined (VALUE_VERSION_KEY, VALUE_INSTANCE_NAME_KEY, ...). Limited to 256 utf-8 encoded Bytes. Keys starting with “apm:” are reserved for internal use.
- `valueName`: The display name of the property (VALUE_VERSION_NAME, VALUE_INSTANCE_NAME_NAME,... for the examples above). Limited to 512 utf-8 encoded Bytes.
- `value`: The value of the property. Limited to 32768 utf-8 encoded Bytes.

Remarks:

Application values may be changed at runtime but must not have a high change frequency and do not have any history of previous values.

3.5.5 Method Unregister (.NET)

The method Unregister will unregister the application and will tell the app.telemetry Library to disconnect from the app.telemetry agent. It is not valid to call further methods on that object after unregistering.

Syntax

```
public void Unregister();
```

3.5.6 Method RegisterFilterValue (.NET)

`RegisterFilterValue` registers application specific filter item, which may be used as filter strings in the `CreateContext` function. Registering the filter values helps the user selecting filters when starting Software-Telemetry sessions in the client interface.

To improve the registration process enumerating the possible filter values, instead of registering the filter values by calling `RegisterFilterValues` right after application registration, you may create a subclass of APMApplication implementing the callback function `EnumAppFilterValuesCallback`. This callback method will be invoked only if no application with same application name, application id and application tier has already registered filter values. Enumerate and register the filter values in the callback function. This mechanism is designed e.g. for web service farms, where the enumeration of the filters will register all users. So it is sufficient to register the filter values once on one web service instance.

Syntax

```
public void RegisterFilterValue(String filterValue,
                                String filterDescription);
```

Parameters:

- `filterValue` (`String`): The exact filter value as a string as used in the CreateContext method. (e.g. the user login string: "`domain\user.name`")
- `filterDescription` (`String`): Description string of the filter. (e.g. a display string for the user login: "full user name")

Return Value:

This function does not return any value.

Remarks:

You should register all filters that are available in your application so that the user can choose in the client interface from a list box.

Usage Example

```
class TestApplication : APMApplication {

    public TestApplication (String appTierId)
           : base("Fabasoft app.telemetry", "Demo", "App1", appTierId)
    {

        // ...
    }

    public override void EnumAppFilterValuesCallback()
    {

        RegisterFilterValue("user1", "displayname1");
        RegisterFilterValue("user2", "displayname2");
        RegisterFilterValue("user3", "displayname3");
        RegisterFilterValue("user4", "displayname4");
        RegisterFilterValue("user5", "displayname5");

    }
}
```

3.5.7 Method CreateContext (.NET)

Call `CreateContext` to start a new request in the Software-Telemetry context.

If the filter value matches a currently started session filter, logging is started for the current execution thread and the request will be associated with the session. You may call `CreateContext` more than once in one request in order to pass additional filter values. The request will start at the first `CreateContext` and will be finished by the first `ReleaseContext`. Make sure to call `ReleaseContext` (see 3.5.8 "*Method ReleaseContext (.NET)*") when exiting the request.

A context can be regarded as a complete request that is traced through several involved threads or modules. And the full context request has a duration that spans the interval from `CreateContext` to `ReleaseContext`.

Syntax

```
public void CreateContext(String filtervalue);
```

Parameters:

- `filtervalue` (`String`): (may be empty) Value to be matched against the session start value.

Return Value:

This function does not return any value.

See chapter 1.7.3 “Application Properties” for more information on the app.telemetry request concept.

Usage Example

```
public static void doRequest(APMApplication app, string sequenceId)
{
    app.CreateContext(FILTER_VALUE);
    /* place you request processing code here ... */
    app.ReleaseContext();
}
```

3.5.8 Method ReleaseContext (.NET)

Call `ReleaseContext` to indicate the end of the processing of one request. One `ReleaseContext` will close the request no matter how many `CreateContext` or `AttachContext` calls have been called.

Syntax

```
public void ReleaseContext();
```

Remarks:

If you don't call `ReleaseContext` the request keeps the running status until the request-timeout has been reached.

Usage Example

```
public static void doRequest(APMApplication app, string sequenceId)
{
    app.CreateContext(FILTER_VALUE);
    /* place you request processing code here ... */
    app.ReleaseContext();
}
```

3.5.9 Method GetContext (.NET)

Call the `GetContext`-method to get a new context handle from the SDK.

This context handle is used to synchronize different threads/processes/modules - this helps you track the control flow through a distributed environment. The acquired context handle has to be submitted (either by transmitting over the network or in another way) to the other thread/process/module which has to call the `AttachContext`-method with the same context.

For more information see also 1.8.1 “*Passing Context*” and 3.5.10 “*Method AttachContext (.NET)*”.

Syntax

```
public Boolean GetContext(out byte[] requestinfo);
```

Parameters:

- `requestinfo`: `GetContext` creates a context at the location pointed to by `requestinfo`.

Return Value:

- `true`: if the function succeeds, the return value is true and a context has been saved to the out parameter `requestinfo`.
- `false`: if no context has been created.

Usage Example

```
public void GetRepositoryInfo()
{
    // ...

    byte[] context;
    app.GetContext(out context);

    request.Headers.Add("x-apm-telemetry-context",
            System.Convert.ToBase64String(context));

    // ...

    HttpWebResponse response = (HttpWebResponse)request.GetResponse();
    String syncMark = response.GetResponseHeader("x-apm-telemetry-
syncmark");

    if (syncMark != null) {
        app.SetSyncMark(Convert.FromBase64String(syncMark));
    }

    // ...
}
```

3.5.10 Method AttachContext (.NET)

Call `AttachContext` to restore the context acquired through `GetContext` in another thread or process.

For more information see also 1.8.1 “*Passing Context*” and 3.5.9 “*Method GetContext (.NET)*”.

Syntax

```
public void AttachContext(byte[] requestinfo);
```

Parameters:

- `requestinfo`: The context which you want to attach to. You get such a context by calling `GetContext`.

Usage Example

```
void ContinueRequest(byte[] requestinfo) {
    m_application.AttachContext(requestinfo);
}
```

3.5.11 Method GetSyncMark (.NET)

Call the `GetSyncMark`-method to get a synchronization handle that can be passed to another thread/process/module to synchronize control flow.

The difference between a s*ync-marks* and a *context* is:

- A *context* is used to associate with one request. It is created by the initiator and attached to by other threads/process/modules belonging to the same request.
- A s*ync-mark* is only a time synchronization handle between the different threads/processes/modules which can be used several times to synchronize the control flow exactly (different systems may have different timestamps and different time precision).

For more information see also 1.8.2 “*Synchronizing Timeline with SyncMarks*”.

Syntax

```
public Boolean GetSyncMark(out byte[] syncMark);
```

Parameters:

- `syncMark`: `GetSyncMark` creates a sync-mark at the location pointed to by `syncMark`.

Return Value:

- `true`: if the function succeeds, the return value is true and a sync-mark has been saved to the out parameter `syncMark`.
- `false`: if no sync-mark has been created.

Usage Example

```
void continueRequest(APMApplication appl)
{
    byte []syncMark;
    Boolean success = app.GetSyncMark(syncMark);
}
```

3.5.12 Method SetSyncMark (.NET)

Call the `SetSyncMark`-method to pass and submit a synchronization handle that you got from the `GetSyncMark`-call to synchronize the sequence of events.

For more information see also 1.8.2 “*Synchronizing Timeline with SyncMarks*”, 3.5.11 “*Method GetSyncMark (.NET)*” and 3.5.9 “*Method GetContext (.NET)*”.

Syntax

```
public void SetSyncMark(byte[] syncMark);
```

Parameters:

- `syncMark`: The sync-mark which you want to sync with.

Return Value:

This function does not return any value.

Usage Example

See 3.5.9 “*Method GetContext (.NET)*”.

3.5.13 Method HasActiveContext (.NET)

The API method `HasActiveContext` defined in the APMApplication class will tell you if there is currently a Software-Telemetry session for the given log level for the application (the one that is instrumented with the APM SDK) running.

This information should be used to avoid costly operations preparing parameters for events that are currently not logged.

Syntax

```
public Boolean HasActiveContext(Byte level);
```

Parameters:

- `Level` (`Byte`): Software-Telemetry log level to be checked.

Return Value:

- `true`: if logging is active for this application for the given log level.
- `false`: otherwise (if logging is not active for the given log level) or if the native Software-Telemetry library is not loaded correctly.

Usage Example

```
private void hasContext_Click(object sender, EventArgs e) {
    Byte level = getEventLevel(hasContextLevel.SelectedIndex);
    hasCtxResult.Text = (app.HasActiveContext(level)) ? "yes" : "no";
}
```

3.5.14 Method Report (.NET)

Call the `Report`-method from the `APMApplication` class to create a new feedback report.

The filter value will be matched to the description using the registered filter values.

Use the textual description of the feedback entered by the user or generated by the application to identify the report session later again.

Syntax

```
public void Report(String filtervalue, String keyvalue, String
description);
```

Parameters:

- `filtervalue (String)`: the raw filter value, which was registered with `RegisterFilterValue`
- `reportkey (String)`: A key value may be generated by the calling application to associate further descriptions to a feedback. If empty, a new feedback report will be generated on each call.
- `description (String)`: Textual description of the feedback entered by the user or generated by the application. The maximum length of the description is **32 768** Bytes.

Return Value:

This function does not return any value.

Usage Example

```
public APMApplication app;

public void RegisterTelemetry {
    app = new APMApplication("appName", "appId", "appTierName",
"appTierId");
}

public void generateReport(String username, String description) {
    app.Report(username, "repkey_1", description);

    app.ReportValue("repkey_1", "username", "John Doe");

    System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
    byte[] var = enc.GetBytes("any text");
    app .ReportContent("repkey_1", "my-text.txt", "text/plain", ref var);

    Stream stream = File.OpenRead(reportFilePath);
    app.ReportFile("repkey_1", "my-image.png", "image/png", stream);
}
```

3.5.15 Method ReportValue (.NET)

Call `ReportValue` to add a key/value pair to an existing report.

For more information see 3.5.14 “*Method Report (.NET)*”.

Syntax

```
public void ReportValue(string reportkey, string key, string value);
```

Parameters:

- `reportkey (String)`: match the reportkey of the `Report` function to associate the value with the report session.
- `key (String)`: A key describes the meaning of the value. Only one value can be associated to a key in one report session.
- `Value (String)`: Textual value (max. 32 kBytes).

Return Value:

This function does not return any value.

Usage Example

`See` 3.5.14 “*Method Report (.NET)*”

3.5.16 Method ReportContent (.NET)

Call `ReportContent` to add content to an existing report.

For more information see 3.5.14 “*Method Report (.NET)*”.

Syntax

```
public void ReportContent(string reportkey, string filename, string
```

```
mimetype, ref byte[] data);
```

Parameters:

- `reportkey (String)`: match the reportkey of the `Report` function to associate the content with the report session.
- `filename (String)`: A filename describing the content of the file. (max. 256 Bytes) – this is the name of the file in the reported session with which the data is persisted.
- `mimetype (String)`: MIME-Type of content – (e.g.: `text/plain`, `image/png`, ...) – later used for presentation of content.
- `data (byte[])`: Byte array containing binary data.

Return Value:

This function does not return any value.

Usage Example

`See` 3.5.14 "*Method Report (.NET)*"

3.5.17 Method ReportFile (.NET)

Call `ReportFile` to add content to an existing report.

For more information see 3.5.14 "*Method Report (.NET)*".

Syntax

```
public void ReportFile(string reportkey, string filename, string
mimetype, System.IO.Stream stream);
```

Parameters:

- `reportkey (String)`: match the reportkey of the `Report` function to associate the content with the report session.
- `filename (String)`: A filename describing the content of the file. (max. 256 Bytes) – this is the name of the file in the reported session with which the data is persisted.
- `mimetype (String)`: MIME-Type of content – (e.g.: `text/plain`, `image/png`, ...) – later used for presentation of content.
- `stream (System.IO.Stream)`: Stream to read the content from. The content will be read in junks of 4 kBytes and transferred to the server.

Return Value:

This function does not return any value.

Usage Example

`See` 3.5.14 "*Method Report (.NET)*"

3.6 APMModule Class and Methods (.NET)

The `APMModule`-class represents a registered Module. You have to register a module to set telemetry points. You may register a module multiple times with the same name. Events of equally named modules will show up in the same column.

3.6.1 Constructor APMModule (.NET)

The constructor registers the module with the given name.

Syntax

```
public APMModule(string name);
```

3.6.2 Method RegisterEvent (.NET)

It is required and important to register the events used in your application in order to get the correct event names and parameter descriptions as provided. The `RegisterEvent`-method will tell the app.telemetry server a name for every event-ID and optionally parameter descriptions for the parameters used in the event calls.

Syntax

```
public void RegisterEvent(UInt64 eventid, String eventDescription, String
parameterDescription);
```

Parameters:

- `eventid (UInt64)`: Unique ID of event which is used later on for every instrumentation point.
- `eventDescription (String)`: Description string of event. A describing name for that event.
- `parameterDescription (String)`: Name of event parameters. If more parameters are used in one event, separate the parameter names by semi-colon (`;`). For example: "`name;count;size`"

Return Value:

This function does not return any value.

Usage Example

```
public void ModuleEvents()
{
  module = new APMModule("mymodule");
  module.RegisterEvent(101, "eventdesc1", "name;count;size");
  module.RegisterEvent(102, "eventdesc2", null);
  module.RegisterEvent(103, "eventdesc3", null);
  module.RegisterEvent(104, "eventdesc4", null);
  module.RegisterEvent(105, "eventdesc5", null);
}
```

3.6.3 Method Event (.NET)

Call the simple `Event`-method to log events without parameters.

Only the event-ID and the flags define the representation of the event that will be displayed with the registered event name for the used id.

This method is part of the APMModule class and has to be called on your own module implementation class.

Syntax

```
public void Event(UInt64 eventid, Byte level, Bytes flags);
```

Parameters:

- `eventid (UInt64)`: id of the event being processed. Should be defined as constant in your module implementation class.
- `level (Byte)`: event detail level for filtering. For the list of available log levels see chapter 3.2.2 “*Log Level (.NET)*”.
- `flags (Byte)`: flags to mark events with Enter-, Leave- and/or Wait-semantic. More than one flags can be added together with a logic OR (`|`). For the list of available flags see chapter 3.2.4 “*Predefined Events (.NET)*”.

Return Value:

This function does not return any value.

Usage Example

```
public void WebShopWebMethod(object sender, EventArgs e) {
    // ...

    m_module.Event(SampleModule.EVENT_PROCESS_REQUEST,
        APM.EVENT_LEVEL_NORMAL, APM.FLAG_ENTER);

    GetRepositoryInfo();
    GetStockServiceInfo();

    m_module.Event(SampleModule.EVENT_PROCESS_REQUEST,
        APM.EVENT_LEVEL_NORMAL, APM.FLAG_LEAVE);

    // ...
}
```

3.6.4 Method EventStr (.NET)

Call the `EventStr`-method to log events with a single textual (string) parameter.

In addition to the event-ID (with the registered event name) and the flags a single string parameter value will be logged.

For more information see 3.6.3 “*Method Event (.NET)*”.

Syntax

```
public void EventStr(UInt64 eventid, Byte level, Bytes flags, String
param);
```

Parameters:

- `eventid (UInt64)`: id of the event being processed. Should be defined as constant in your module implementation class.
- `level (Byte)`: event detail level for filtering. For the list of available log levels see chapter 3.2.2 “*Log Level (.NET)*”.
- `flags (Byte)`: flags to mark events with Enter-, Leave- and/or Wait-semantic. More than one flags can be added together with a logic OR (`|`). For the list of available flags see chapter 3.2.4 “*Predefined Events (.NET)*”.
- `param (String)`: a string parameter to be logged – can be any textual or preformatted message.

Return Value:

This function does not return any value.

Usage Example

```
public void WebShopWebMethod(object sender, EventArgs e) {
    // ...

    m_module.EventStr(SampleModule.EVENT_PROCESS_REQUEST,
        APM.EVENT_LEVEL_NORMAL, APM.FLAG_ENTER, "enter web-method");

    GetRepositoryInfo();
    GetStockServiceInfo();

    m_module.EventStr(SampleModule.EVENT_PROCESS_REQUEST,
        APM.EVENT_LEVEL_NORMAL, APM.FLAG_LEAVE, "leave web-method");

    // ...
}
```

3.6.5 Method EventParam (.NET)

Call the `EventParam`-method to log events with a generic parameter structure - with any number and combination of textual (string) values and integer values (32- and 64-bit).

To use the `EventParam`-method you have to allocate a new `APMParameter`. With a few helper methods you can add values to this parameter object and finally you can use the parameter object in the event function to be logged. For details how to allocate and fill the `APMParameter` object see chapter 3.7 "*APMParameter Class and Methods (.NET)*".

Syntax

```
public void EventParam(UInt64 eventid, Byte level, Byte flags,
APMParameter param);
```

Parameters:

- `eventid (UInt64)`: id of the event being processed. Should be defined as constant in your module implementation class.
- `level (Byte)`: event detail level for filtering. For the list of available log levels see chapter 3.2.2 "*Log Level (.NET)*".
- `flags (Byte)`: flags to mark events with Enter-, Leave- and/or Wait-semantic. More than one flags can be added together with a logic OR (`|`). For the list of available flags see chapter 3.2.4 "*Predefined Events (.NET)*".
- `param (APMParameter)`: the parameter object containing any number of values to be logged.

Return Value:

This function does not return any value.

Usage Example

```
public void WebShopWebMethod(object sender, EventArgs e) {
    // ...

    APMParameter param = new APMParameter();

    param.AddInt32(75);
    param.AddInt64(2147483698002);
    param.AddString("enter WebShop web-method");
```

```
    module.EventParam(SampleModule.EVENT_PROCESS_REQUEST,
        APM.EVENT_LEVEL_NORMAL, APM.FLAG_ENTER, param);
    // ...
}
```

3.7 APMParameter Class and Methods (.NET)

The `APMParameter`-class is an object structure for storing and managing parameter values that can be used for logging complex value structures with more than one or different type of values.

See also 3.6.5 “*Method EventParam (.NET)*”.

3.7.1 Constructor APMParameter (.NET)

Call the `APMParameter`-constructor to allocate a new parameter object where you can add any number of textual (string) values and integer values (32- and 64-bit).

Syntax

```
public APMParameter();
```

3.7.2 Method AddInt32 (.NET)

Call the method `AddInt32` on your `APMParameter`-instance to add a simple integer value (32-bit) to that object.

Syntax

```
public void AddInt32(UInt32 value);
```

Parameters:

- `value (UInt32)`: 32-bit integer value to be added to the parameter object.

Return Value:

This function does not return any value.

Usage Example

`See` 3.6.5 “*Method EventParam (.NET)*”.

3.7.3 Method AddInt64 (.NET)

Call the method `AddInt64` on your `APMParameter`-instance to add a simple integer value (64-bit) to that object.

Syntax

```
public void AddInt64(UInt64 value);
```

Parameters:

- `value (UInt64)`: 64-bit integer value to be added to the parameter object.

Return Value:

This function does not return any value.

Usage Example

`See` 3.6.5 “*Method EventParam (.NET)*”.

3.7.4 Method AddString (.NET)

Call the method `AddString` on your `APMParameter`-instance to add a textual (string) value to that object.

Syntax

```
public void AddString(String value);
```

Parameters:

- `value (String)`: textual parameter value to be added to the parameter object.

Return Value:

This function does not return any value.

Usage Example

`See` 3.6.5 “*Method EventParam (.NET)*”.

4 Java API Reference

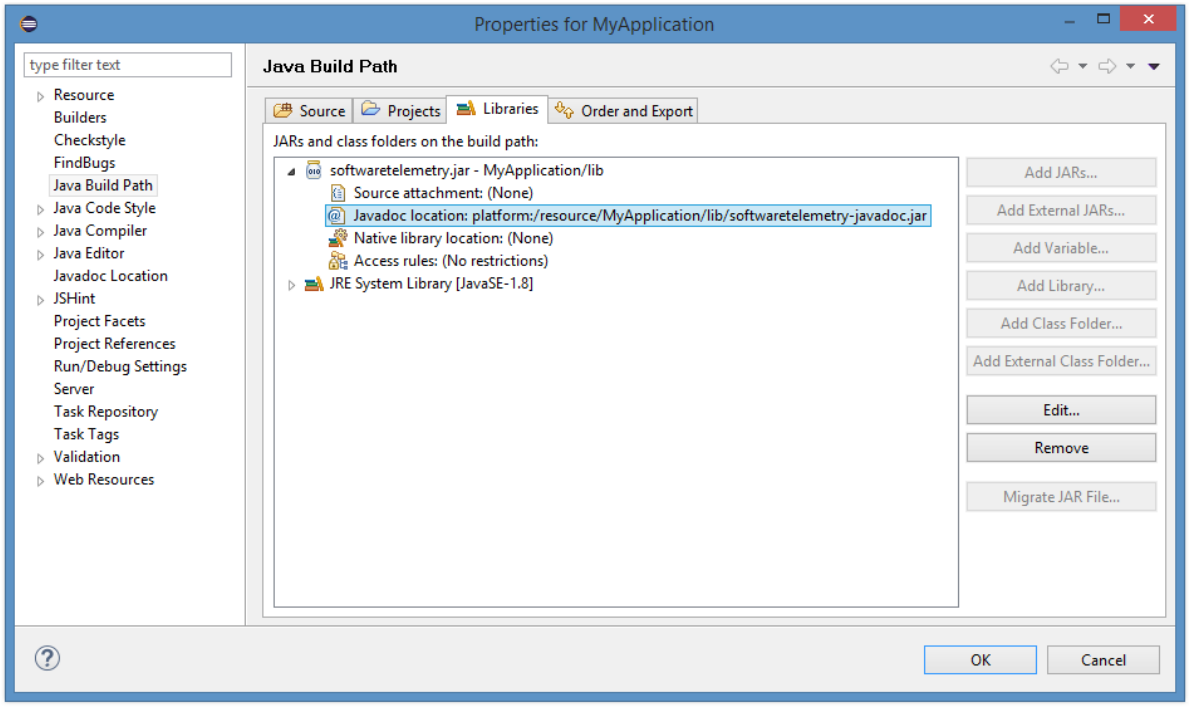
The Fabasoft app.telemetry Java API allows you to instrument any Java application with instrumentation points to see what's going on in complex applications.

In order to use the Software-Telemetry Java API effectively you should use the Eclipse IDE for development and set up an Eclipse Java project as described in the following chapter.

4.1 Configure Eclipse for Java Project

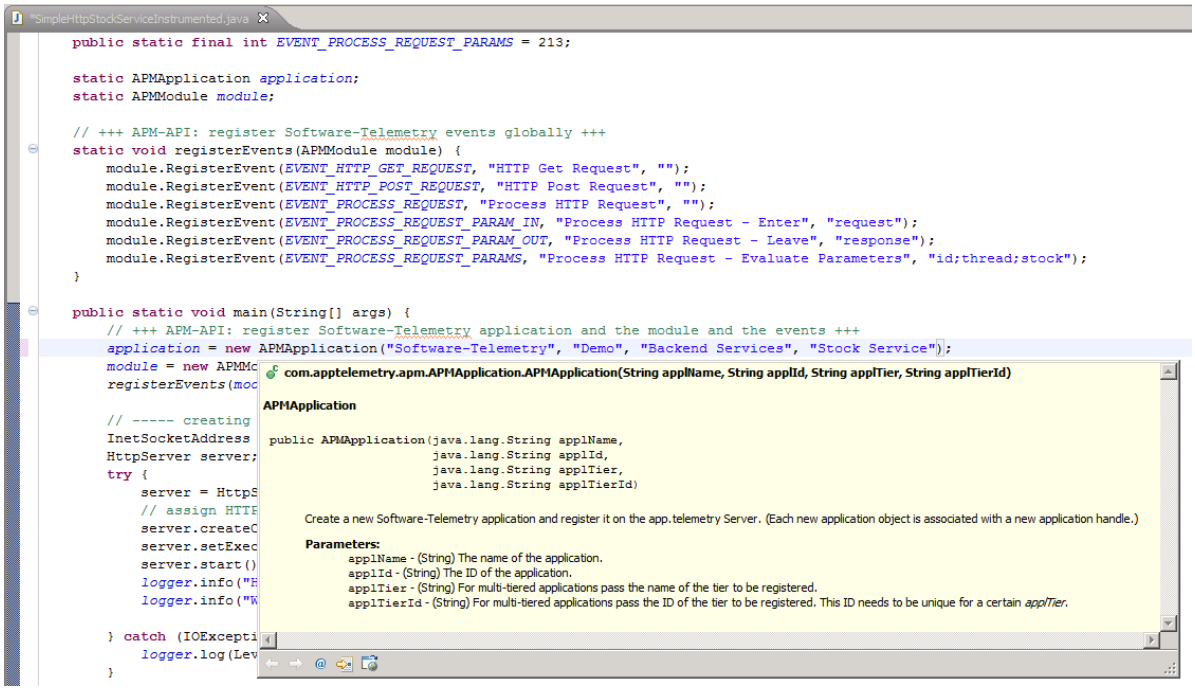
For your new Eclipse Java project you have to include the following Fabasoft app.telemetry SDK resources:

- The Software-Telemetry library "`softwaretelemetry.jar`" as referenced library. This Java library is a wrapper for the native binary library "`softwaretelemetry-java64.dll`" delivered with the Fabasoft app.telemetry agent.
- The Javadoc for this library "`softwaretelemetry-javadoc.jar`". This file should be set as *Javadoc-location* for the `softwaretelemetry.jar` library.



On the same server as your Java project is located, you have to have the Fabasoft app.telemetry agent installed, which includes the binary library "`softwaretelemetry-java64.dll`" and handles the native telemetry data processing.

A correct configured Eclipse project provides *ContentAssist* and *Javadoc* help for efficient development of an instrumented software solution.



The Java Software-Telemetry SDK is packaged all together into the Java package: `com.apptelemetry.apm` where the basic classes and the constants are defined.

4.2 Constants (Java)

All the constants are static members of the *APM* class and can be accessed in a static way.

4.2.1 Flags (Java)

Flags are used to specify the type of an event and are passed as argument to every event call. They are predefined by the API on the base class `APM` with the data type `Byte` and have the following possible values:

- `FLAG_NONE`
- `FLAG_ENTER`
- `FLAG_LEAVE`
- `FLAG_WAIT`
- `FLAG_WARNING`
- `FLAG_ERROR`
- `FLAG_PROCESS_INFO`

For more details about the flags and there meaning see the general chapter 1.7.1 “*Flags*”.

4.2.2 Log Level (Java)

An event will only be logged if the event-level is lower or equal to the value selected in the session or log-definition. The log levels are predefined by the API on the base class `APM` with the data type `Byte` and have the following possible values:

- `EVENT_LEVEL_LOG`

- `EVENT_LEVEL_IPC` (only internal)
- `EVENT_LEVEL_NORMAL`
- `EVENT_LEVEL_DETAIL`
- `EVENT_LEVEL_DEBUG`

For more details about the log levels and there meaning see the general chapter 1.7.2 “*Log Level*”.

4.2.3 Predefined Modules (Java)

To use a predefined module create a new `APMModule` object (by means of calling the constructor) with the predefined module name. This object will be used later to fire events for that module.

For a list of available standard modules with a set of predefined events see chapter 1.5 “*Standard Modules*”.

Usage Example

```
APMModule mod1 = new APMModule(APM.MODULE_NAME_OS);
APMModule mod2 = new APMModule(APM.MODULE_NAME_XMLHTTP);
```

4.2.4 Predefined Events (Java)

The SDK already provides a set of predefined events used for general purpose. For a full listing of all available predefined events see the generic chapter 1.5.3 “*HttpRequest*

The `HttpRequest` Module is used to trace the parameters of http requests such as URL, method, status, header fields and message bodies.

The constant names are prefixed with “`APM_`” for the C/C++ API (`APM_<constant>`) and in the other object-oriented languages they are members of the `APM` base class (`APM.<constant>`).

Name	Value	Eventname	Parameters
`APM_MODULE_NAME_HTTP` `APM.MODULE_NAME_HTTP`	"HttpRequest"		
`APM_EVENT_HTTP_URL` `APM.EVENT_HTTP_URL`	`2000`	"URL"	“scheme;host;path;extra”
`APM_EVENT_HTTP_METHOD` `APM.EVENT_HTTP_METHOD`	`2001`	"Method"	“method”
`APM_EVENT_HTTP_STATUS` `APM.EVENT_HTTP_STATUS`	`2002`	"Status"	“code;message”
`APM_EVENT_HTTP_AUTHENTICATION` `APM.EVENT_HTTP_AUTHENTICATION`	`2003`	"Authentication"	“type;credential”
`APM_EVENT_HTTP_PROXY` `APM.EVENT_HTTP_PROXY`	`2004`	"Proxy"	“server;username”
`APM_EVENT_HTTP_REQUEST_HEADER`	`2010`	"Request Header"	“header;value”

APM.EVENT_HTTP_REQUEST_HEADER			
APM_EVENT_HTTP_REQUEST_CONTENT TYPE APM.EVENT_HTTP_REQUEST_CONTENT TYPE	2011	"Request Content Type"	“media-type;charset;boundary”
APM_EVENT_HTTP_REQUEST_CONTENT BODY APM.EVENT_HTTP_REQUEST_CONTENT BODY	2012	"Request Body Part"	
APM_EVENT_HTTP_RESPONSE_HEADER APM.EVENT_HTTP_RESPONSE_HEADER	2020	“Response Header”	“header;value”
APM_EVENT_HTTP_RESPONSE_CONTEN TTYPE APM.EVENT_HTTP_RESPONSE_CONTEN TTYPE	2021	"Response Content Type"	“media-type;charset;boundary”
APM_EVENT_HTTP_RESPONSE _CONTENTBODY APM.EVENT_HTTP_RESPONSE _CONTENTBODY	2022	"Response Body Part"	

Predefined Events”.

All predefined events are declared and available as global constants on the base class “APM”.

Error/Trace Events:

Use these standard events to mark error, warning or info conditions in your application:

- EVENT_ERROR
- EVENT_WARNING
- EVENT_INFO
- EVENT_TRACE

4.2.5 Predefined Application Property Keys and Names (Java)

View the language agnostic Application Properties section for an explanation of the properties listed below.

All predefined application property keys and names are available as public constants on the APM class.

- PROPERTY_INSTANCE_GUID_KEY with name PROPERTY_INSTANCE_GUID_NAME

4.2.6 Predefined Application Value Keys and Names (Java)

View the language agnostic Application Values section for an explanation of the properties listed below.

All predefined application value keys and names are available as public constants on the APM class.

- `VALUE_VERSION_KEY` with name `VALUE_VERSION_NAME`
- `VALUE_INSTANCE_NAME_KEY` with name `VALUE_INSTANCE_NAME_NAME`

4.3 Data Types / Class Overview (Java)

4.3.1 APM (Java)

The `APM` class is the base class of the Software-Telemetry SDK. It contains all global defined constants as described in chapter 4.2 "*Constants (Java)*". Use the methods from the `APM` class in a static way (there is no new instance allocation required).

The `APM` class supports the following global static methods which do not require an application scope or status:

- `IsConnected`
- `IsCompatible`
- `DetachThread`
- `AttachThread`

A detailed description of this class with all methods explained is described in chapter 4.4 "*APM Class and Methods (Java)*".

4.3.2 APMApplication (Java)

`APMApplication` is the base class for own application implementation and handles the request scope and reporting functions.

After creating an instance with the class constructor passing the application registration information, it stores the application handle inside and provides all context-specific functions for your application.

The `APMApplication` class supports the following instance/class methods which require an application scope and have to be called on your object instance of `APMApplication`:

- `RegisterFilterValue`
- `CreateContext`
- `AttachContext`
- `ReleaseContext`
- `GetContext`
- `GetSyncMark`
- `SetSyncMark`
- `HasActiveContext`
- `Report`
- `ReportValue`
- `ReportContent`

A detailed description of this class with all methods explained is described in chapter 4.5 "*APMApplication Class and Methods (Java)*".

Note: At the end of your program lifecycle you should call the `Unregister()`-function to be sure that also the native library (JNI) objects are released (to prevent memory leaks).

4.3.3 APMModule (Java)

`APMModule` is the base class for own module implementation and is used to fire instrumentation events.

After creating an instance with the class constructor passing the module name, it stores the module handle inside and provides all module-specific functions for your application.

The `APMModule` class supports the following instance/class methods which require a module scope and have to be called on your object instance of `APMModule`:

- `RegisterEvent`
- `Event`
- `EventStr`
- `EventParam`

A detailed description of this class with all methods explained is described in chapter 4.6 "*APMModule Class and Methods (Java)*".

Note: At the end of your program lifecycle you should call the `Unregister()`-function to be sure that also the native library (JNI) objects are released (to prevent memory leaks).

4.3.4 APMParameter (Java)

`APMParameter` is used as data class for generic event parameters.

To define any parameters used in events you have to create a new `APMParameter` instance and add the typed parameters to that instance:

- `AddInt32`
- `AddInt64`
- `AddString`

A detailed description of this class with all methods explained is described in chapter 4.7 "*APMParameter Class and Methods (Java)*".

4.3.5 APMCompatibilityInfo (Java)

The `APMCompatibilityInfo` class is a simple data structure providing the state and the message of the library compatibility.

The current `APMCompatibilityInfo` can be obtained using the public method call: `APM.IsCompatible()` which will return an instance of the class `APMCompatibilityInfo` (see 4.4.2 "*Method IsCompatible (Java)*").

Type Declaration

```
package com.apptelemetry.apm;

public class APMCompatibilityInfo {

    public boolean compatible;
    public String message;

}
```

4.4 APM Class and Methods (Java)

The `APM` class is the base class of the Software-Telemetry SDK. It contains all global defined constants as described in chapter 4.2 "*Constants (Java)*". Use the methods from the `APM` class in a static way (there is no new instance allocation required).

Remarks:

With version 2010 Fall Release the application-specific functions (`CreateContext`, ..., `Report`, ...) of the Java API where moved to the `APMApplication` class (see chapter 4.5 "*APMApplication Class and Methods (Java)*") and have to be called on an object instance of this class!

4.4.1 Method IsConnected (Java)

The API method `IsConnected` will tell you if the application (the one that is instrumented with the APM SDK) has successfully registered to an app.telemetry agent and is currently connected to the agent process.

Only when an application is successfully connected to an app.telemetry agent, Software-Telemetry data will be processed and sent to and collected by the app.telemetry server

Syntax

```
public static boolean IsConnected();
```

Parameters:

This function does not take any arguments.

Return Value:

- `true`: if the application has successfully registered to and connected to an app.telemetry agent.
- `false`: if the application has not registered or is currently not connected to an app.telemetry agent or if the native Software-Telemetry library is not loaded correctly.

Usage Example

```
public static void main(String[] args) {
    boolean apiConnected = APM.IsConnected();
    System.out.println("Library connected via agent to server: " +
apiConnected);
}
```

4.4.2 Method IsCompatible (Java)

Call `IsCompatible` to check if the native library is compatible with the implementation in the current `softwaretelemetry.jar` archive.

Syntax

```
public static APMCompatibilityInfo IsCompatible();
```

Parameters:

This function does not take any arguments.

Return Value:

- Instance of `APMCompatibilityInfo`. The member `compatible` is set to true if the library is compatible. The `message` member contains a descriptive text and an optional product name and version number.

Remarks:

For details about the object structure of `APMCompatibilityInfo` see chapter 4.3.5 "*APMCompatibilityInfo (Java)*"

Usage Example

```
public static void main(String args[]) {
    APMCompatibilityInfo ci = APM.IsCompatible();
    if (!ci.compatible) {
        System.out.println("INCOMPATIBLE: " + ci.message);
        System.exit(1);
    } else {
        System.out.println(ci.message);
    }
}
```

4.4.3 Method DetachThread (Java)

Call the `DetachThread`-method when processing of a particular request is suspended and will be resumed later in this or another thread in the same process. Pass the handle returned by this method to the `AttachThread`-method to resume processing (see 4.4.4 "*Method AttachThread (Java)*").

This functionality is in particular useful in queuing systems, where a large number of requests will be queued and processed by a small number of worker threads asynchronous to the initial request. Using the *Detach-/AttachThread* methods you can follow the request through the process and even measure the time spent in the queue, even if a different thread continues processing.

Syntax

```
public static long DetachThread()
```

Parameters:

This function does not take any arguments.

Return Value:

- `DetachThread` generates an id (64bit) that uniquely identifies the current state of the request. If the function succeeds, the return value is not 0.
- The function will return `0` in case the library is not available or if no request has been started using `CreateContext` and/or `AttachContext`.

Usage Example

```
public class Task {
    //...
    public long tx;
```

```
    //...
    public void Initialize(/* ... */)
    {
        app.CreateContext("filterstring");
        module.Event(EVENT_ENTER_TASK_TO_QUEUE, APM.EVENT_LEVEL_NORMAL,
APM.FLAG_ENTER);
        tx = APM.DetachThread();
        schedule();
    }
    public void process()
    {
        APM.AttachThread(tx);
        module.Event(EVENT_REMOVE_TASK_FROM_QUEUE,
APM.EVENT_LEVEL_NORMAL, APM.FLAG_LEAVE);
        // process work
        app.ReleaseContext();
    }
}
```

Remarks:

Be careful to use `DetachThread` and `AttachThread` in a balanced way. `DetachThread` will store the current state of the request in memory and `AttachThread` will remove the information from memory while restoring the request state. So calls to `DetachThread` lacking a continuation will leak memory and will leave the request in a "*running*" state. Multiple calls to `AttachThread` will not have an effect, because the request state is only available to the first call to `AttachThread`.

4.4.4 Method AttachThread (Java)

Call the `AttachThread`-method when processing of a particular request is resumed after being suspended in this or another thread of the same process. Pass the handle returned by `DetachThread`-method to restore the recorded state of the request.

For more details see 4.4.3 “*Method DetachThread (Java)*”.

Syntax

```
public static void AttachThread(long transaction);
```

Parameters:

- `transaction`: AttachThread uses the transaction handle generated by `DetachThread` to identify the current state of the request.

Return Value:

This function does not return any value.

Usage Example

`See` 4.4.3 “*Method DetachThread (Java)*”

4.5 APMApplication Class and Methods (Java)

The `APMApplication` class is the base class for your application registration, the context- and report-functions and the filter registration (callback or directly). It stores the application handle inside and provides all application-specific functions for your application to be called on your `APMApplication` object instance.

Your own application implementation may override the callback function `EnumFilterValuesCallback` (default implementation is empty) to provide the filters values via the callback function. Alternatively the filters may also be registered directly.

The application is registered when a new instance of the APMApplication class is being created.

Note: At the end of your program lifecycle you should call the `Unregister()`-function to be sure that also the native library (JNI) objects are released (to prevent memory leaks).

4.5.1 Constructor APMApplication (Java)

Syntax

```
public APMApplication(String appName, String appId, String appTierName,
String appTierId);
```

Parameters:

- `appName`: The name of your application
- `appId`: The id of your application
- `appTierName`: The tier inside the application
- `appTierId`: id to distinguish multiple services of one application tier

Usage Example

```
public static void main(String args[]) {

    APMApplication app = new APMApplication("Java Test-App", "1", "Tier",
"1");
    APMModule mod = new APMModule("module1");

    registerEvents(mod);
    registerFilters(app);

    app.CreateContext("MyFilter");

    mod.Event(10, APM.EVENT_LEVEL_LOG, APM.FLAG_ENTER);
    // ... process request and fire more events
    mod.Event(10, APM.EVENT_LEVEL_LOG, APM.FLAG_LEAVE);

    app.ReleaseContext();

    mod.Unregister();
    app.Unregister();

}
```

4.5.2 Method RegisterApplicationProperty (Java)

Call RegisterApplicationProperty to register additional application registration properties immediately after you call to the APMApplication class constructor to ensure proper operation.

Syntax

```
public void RegisterApplicationProperty(String propertyKey, String
propertyName, String propertyValue);
```

Parameters:

- `propertyKey`: The application defined key of the property or the predefined PROPERTY_INSTANCE_GUID_KEY. Limited to 256 utf-8 encoded Bytes. Keys starting with “apm:” are reserved for internal use.
- `propertyName`: The display name of the property (PROPERTY_INSTANCE_GUID_NAME for the example above). Limited to 512 utf-8 encoded Bytes.
- `propertyValue`: The value of the property. Limited to 32768 utf-8 encoded Bytes.

Remarks:

Application registration Properties cannot be changed after they have been set.

4.5.3 Method RegisterApplicationValue (Java)

Syntax

```
public void RegisterApplicationValue(String valueKey, String valueName,
String value);
```

Parameters:

- `valueKey`: The application defined key of the value or one of the predefined (VALUE_VERSION_KEY, VALUE_INSTANCE_NAME_KEY, ...). Limited to 256 utf-8 encoded Bytes. Keys starting with “apm:” are reserved for internal use.
- `valueName`: The display name of the property (VALUE_VERSION_NAME, VALUE_INSTANCE_NAME_NAME,... for the examples above). Limited to 512 utf-8 encoded Bytes.
- `value`: The value of the property. Limited to 32768 utf-8 encoded Bytes.

Remarks:

Application values may be changed at runtime but must not have a high change frequency and do not have any history of previous values.

4.5.4 Method Unregister (Java)

Call `Unregister` at the end of your program to tell the Fabasoft app.telemetry agent that your program lifetime ends now.

The automatic unregister call via the object destructor may not work properly because the Java native library (JNI) also keeps a reference on the object which may prevent the destructor from run after the object is not used any longer in your application code.

The explicit call of `Unregister` will release any references to the `APMApplication` also in the Java native library (JNI).

Syntax

```
public void Unregister();
```

4.5.5 Method RegisterFilterValue (Java)

`RegisterFilterValue` registers application specific filter items, which may be used as filter strings in the `CreateContext` function. Registering the filter values helps the user selecting filters when starting Software-Telemetry sessions in the client interface.

There are two possibilities how to register the filter values for your application:

- *Directly at program start*: you have to provide all filter values after your application is created/registered - by calling the `registerFilterValue` method with all possible filter values.
- *via callback function*: you can implement your own `APMApplication` subclass and implement the callback function `EnumFilterValuesCallback` - where you provide the available filters for your application on invocation by calling the `RegisterFilterValue` method.

The `APMApplication` class is not abstract but provides an empty method body for the callback function `EnumFilterValuesCallback`.

This callback method will be invoked only if no application with same application name, application id and application tier has already registered filter values. Enumerate and register the filter values in the callback function. This mechanism is designed e.g. for web service farms, where the enumeration of the filters will register all users. So it is sufficient to register the filter values once on one web service instance.

Syntax

```
public void RegisterFilterValue(String filterValue, String
filterDescription);
```

Parameters:

- `filterValue` (`String`): The exact filter value as a string as used in the CreateContext method. (e.g. the user login string: "`domain\user.name`")
- `filterDescription` (`String`): Description string of the filter. (e.g. a display string for the user login: "full user name")

Return Value:

This function does not return any value.

Remarks:

You should register all filters that are available in your application so that the user can choose in the client interface from a list box.

Usage Example

```
// Extend the APMApplication API class with your own implementation
//    to provide the events and filters of your application.

class InstrumentedApp extends APMApplication {

    // own constructor invoking super constructor from base class
    public InstrumentedApp(String appName, String appId, String
appTierName, String appTierId) {
        super(appName, appId, appTierName, appTierId);
    }

    public void EnumFilterValuesCallback() {
        RegisterFilterValue("user.name1", "Test User 1");
        RegisterFilterValue("user.name2", "Test User 2");
        RegisterFilterValue("user.name3", "Test User 3");
```

```
        // or get a map of all used login names with description
        // and add them by means of iterating over all entries.
    }
} // InstrumentedApp
```

4.5.6 Method CreateContext (Java)

Call `CreateContext` to start a new request in the Software-Telemetry context.

If the filter value matches a currently started session filter, logging is started for the current execution thread and the request will be associated with the session. You may call `CreateContext` more than once in one request in order to pass additional filter values. The request will start at the first `CreateContext` and will be finished by the first `ReleaseContext`. Make sure to call `ReleaseContext` (see 4.5.7 "*Method ReleaseContext (Java)*") when exiting the request.

A context can be regarded as a complete request that is traced through several involved threads or modules. And the full context request has a duration that spans the interval from `CreateContext` to `ReleaseContext`.

Syntax

```
public void CreateContext(String filtervalue);
```

Parameters:

- `filtervalue` (`String`): (may be empty) Value to be matched against the session start value.

Return Value:

This function does not return any value.

See chapter 1.7.3 "*Application Properties*" for more information on the app.telemetry request concept.

Usage Example

```
public static void main(String args[]) {
    // ...
    APMApplication app = new APMApplication(...);
    app.CreateContext("filtervalue");
    // ...
    // do anything with the Software-Telemetry (log events, ...)
    // ...
    app.ReleaseContext();
    // ...
}
```

4.5.7 Method ReleaseContext (Java)

Call `ReleaseContext` to indicate the end of the processing of one request. One `ReleaseContext` will close the request no matter how many `CreateContext` or `AttachContext` calls have been called.

Syntax

```
public void ReleaseContext();
```

Remarks:

If you don't call `ReleaseContext` the request keeps the running status until the request-timeout has been reached.

Usage Example

`See` 4.5.6 "*Method CreateContext (Java)*"

4.5.8 Method GetContext (Java)

Call the `GetContext`-method to get a new context handle from the SDK.

This context handle is used to synchronize different threads/processes/modules - this helps you track the control flow through a distributed environment. The acquired context handle has to be submitted (either by transmitting over the network or in another way) to the other thread/process/module which has to call the `AttachContext`-method with the same context.

For more information see also 1.8.1 "*Passing Context*" and 4.5.9 "*Method AttachContext (Java)*".

Syntax

```
public byte[] GetContext();
```

Parameters:

This function does not take any arguments.

Return Value:

- This function will return a new context handle as `byte-array`
- or `null` if the native library is not loaded successfully.

Usage Example

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import com.apptelemetry.apm.*;

class InterfaceTestHelpers {
  static void registerFilters(APMApplication application) {
    application.RegisterFilterValue("filter1", "description1");
    // ...
  }

  static void registerEvents(APMModule module) {
    module.RegisterEvent(101, "eventdesc1", null);
    // ...
    module.RegisterEvent(121, "eventParams1", "int32;int64;String");
  }
}


class InterfaceTestRunnable implements Runnable {
  public final Lock lock = new ReentrantLock();
  public byte[] m_syncMark = null;
  public byte[] m_context = null;
  private APMApplication m_application;
  private APMModule m_module;
```

```
  public InterfaceTestRunnable(APMApplication app) {
    m_application = app;
  }

  public void run(){
    lock.lock();
    // create modules and register filters and events (application
already created outside)

    if (m_context == null) {
      m_module = new APMModule("JavaTestService1");
      // sleep 5sec
      m_application.CreateContext("filter1");

    } else {
      m_module = new APMModule("JavaTestService2");
      // sleep 5sec
      m_application.AttachContext(m_context);
    }

    InterfaceTestHelpers.registerEvents(m_module);
    InterfaceTestHelpers.registerFilters(m_application);

    m_module.EventStr(101, APM.EVENT_LEVEL_DEBUG, APM.FLAG_NONE, "start
of the request");

    if (m_context == null) {
      m_context = m_application.GetContext();

    } else {
      m_syncMark = m_application.GetSyncMark();
      module.EventStr(101, APM.EVENT_LEVEL_DEBUG, APM.FLAG_NONE, "end of
the request 2");
      m_application.ReleaseContext();
      lock.unlock();
      return;
    }

    lock.unlock();

    // sleep 3sec

    lock.lock();
    lock.unlock();

    m_application.SetSyncMark(m_syncMark);

    m_module.EventStr(101, APM.EVENT_LEVEL_DEBUG, APM.FLAG_NONE, "end of
the request 1");

    m_application.ReleaseContext();

  }
}


public class InterfaceTestJava {

  public static void main(String args[]) {
    APMApplication app = new APMApplication("InterfaceTestJava
Application", appId, "testAppTierNameJava", "testAppTierIdJava");

    // sleep 1sec
    InterfaceTestRunnable application1 = new InterfaceTestRunnable();
```

```
    // start 2 parallel application threads
    //  (that will communicate with each other and pass the context)
    new Thread(app).start();
    // sleep 1sec
    new Thread(app).start();
  }
}
```

4.5.9 Method AttachContext (Java)

Call `AttachContext` to restore the context acquired through `GetContext` in another thread or process.

For more information see also 1.8.1 “*Passing Context*” and 4.5.8 “*Method GetContext (Java)*”.

Syntax

```
public void AttachContext(byte[] context);
```

Parameters:

- `context`: The context which you want to attach to. You get such a context by calling `GetContext`.

Return Value:

This function does not return any value.

Usage Example

`See` 4.5.8 “*Method GetContext (Java)*”

4.5.10 Method GetSyncMark (Java)

Call the `GetSyncMark`-method to get a synchronization handle that can be passed to another thread/process/module to synchronize control flow.

The difference between a s*ync-marks* and a *context* is:

- A *context* is used to associate with one request. It is created by the initiator and attached to by other threads/process/modules belonging to the same request.
- A s*ync-mark* is only a time synchronization handle between the different threads/processes/modules which can be used several times to synchronize the control flow exactly (different systems may have different timestamps and different time precision).

For more information see also 1.8.2 “*Synchronizing Timeline with SyncMarks*”.

Syntax

```
public byte[] GetSyncMark();
```

Parameters:

This function does not take any arguments.

Return Value:

- This function will return a new synchronization handle as `byte-array`

- or `null` if the native library is not loaded successfully.

Usage Example

`See` 4.5.8 "*Method GetContext (Java)*"

4.5.11 Method SetSyncMark (Java)

Call the `SetSyncMark`-method to pass and submit a synchronization handle that you got from the `GetSyncMark`-call to synchronize the sequence of events.

For more information see also 1.8.2 "*Synchronizing Timeline with SyncMarks*", 4.5.10 "*Method GetSyncMark (Java)*" and 4.5.8 "*Method GetContext (Java)*".

Syntax

```
public void SetSyncMark(byte[] syncMark);
```

Parameters:

- `syncMark`: The synchronization handle that was acquired by `GetSyncMark` from another thread/process/module.

Return Value:

This function does not return any value.

Usage Example

`See` 4.5.8 "*Method GetContext (Java)*"

4.5.12 Method HasActiveContext (Java)

The API method `HasActiveContext` defined in the APMApplication class will tell you if there is currently a Software-Telemetry session for the given log level for the application (the one that is instrumented with the APM SDK) running.

This information should be used to avoid costly operations preparing parameters for events that are currently not logged.

Syntax

```
public boolean HasActiveContext(byte level);
```

Parameters:

- `level` (`byte`): Software-Telemetry log level to be checked.

Return Value:

- `true`: if logging is active for this application for the given log level.
- `false`: otherwise (if logging is not active for the given log level) or if the native Software-Telemetry library is not loaded correctly.

Remarks:

If the native Software-Telemetry library could not be loaded within your Java process a log statement will be sent to the Java logging facility with log level `FINE` and namespace `com.apptelemetry.apm.APM`.

Usage Example

```
public static void main(String[] args) {
    // create application and module and context
    // ... application logic

    if (app.HasActiveContext(APM.EVENT_LEVEL_DETAIL) {
        // do complex preparation of log message
        String logmsg = calcProgressInfo(...);
        // time-consuming operation - do only if telemetry active

        module.EventStr(123, APM.EVENT_LEVEL_DETAIL, APM.FLAG_NONE,
logmsg);
    }

    // finish application logic and release context
}
```

4.5.13 Method Report (Java)

Call the `Report`-method from the `APMApplication` class to create a new feedback report.

The filter value will be matched to the description using the registered filter values.

Use the textual description of the feedback entered by the user or generated by the application to identify the report session later again.

Syntax

```
public void Report(String filtervalue, String keyvalue, String
description);
```

Parameters:

- `filtervalue (String)`: the raw filter value, which was registered with `RegisterFilterValue`
- `reportkey (String)`: A key value may be generated by the calling application to associate further descriptions to a feedback. If empty, a new feedback report will be generated on each call.
- `description (String)`: Textual description of the feedback entered by the user or generated by the application. The maximum length of the description is **32 768** Bytes.

Return Value:

This function does not return any value.

Usage Example

```
public void generateReport(String username, String description, String
reportkey) {

    app.Report(username, reportkey, description);

    app.ReportValue(reportkey, "username", username);

    // read file content into byte array and pass it to report function
    File file = new File("document.html");
    FileInputStream fileInputStream = new FileInputStream(file);
    byte[] data = new byte[(int) file.length()];
    fileInputStream.read(data);
```

```
    fileInputStream.close();
    app.ReportContent(reportkey, "document.html", "text/html", data);
}
```

4.5.14 Method ReportValue (Java)

Call `ReportValue` to add a key/value pair to an existing report.

For more information see 4.5.13 "*Method Report (Java)*".

Syntax

```
public void ReportValue(String reportkey, String key, String value);
```

Parameters:

- `reportkey (String)`: match the reportkey of the `Report` function to associate the value with the report session.
- `key (String)`: A key describes the meaning of the value. Only one value can be associated to a key in one report session.
- `value (String)`: Textual value (max. 32 kBytes).

Return Value:

This function does not return any value.

Usage Example

`See` 4.5.13 "*Method Report (Java)*"

4.5.15 Method ReportContent (Java)

Call `ReportContent` to add content to an existing report.

For more information see 4.5.13 "*Method Report (Java)*".

Syntax

```
public void ReportContent(String reportkey, String filename, String
mimeType, byte[] content);
```

Parameters:

- `reportkey (String)`: match the reportkey of the `Report` function to associate the content with the report session.
- `filename (String)`: A filename describing the content of the file. (max. 256 Bytes) – this is the name of the file in the reported session with which the data is persisted.
- `mimeType (String)`: MIME-Type of content – (e.g.: `text/plain`, `image/png`, ...) – later used for presentation of content.
- `content (byte[])`: Byte array containing binary data.

Return Value:

This function does not return any value.

Usage Example

`See` 4.5.13 "*Method Report (Java)*"

4.6 APMModule Class and Methods (Java)

The `APMModule`-class represents a registered Module. You have to register a module to set telemetry points. You may register a module multiple times with the same name. Events of equally named modules will show up in the same column.

Note: At the end of your program lifecycle you should call the `Unregister()`-function to be sure that also the native library (JNI) objects are released (to prevent memory leaks).

4.6.1 Constructor APMModule (Java)

The constructor registers the module with the given name directly on app.telemetry agent/server.

Syntax

```
public APMModule(string name);
```

4.6.2 Method Unregister (Java)

Call `Unregister` at the end of your program to tell the Fabasoft app.telemetry agent that your program lifetime ends now.

The automatic unregister call via the object destructor may not work properly because the Java native library (JNI) also keeps a reference on the object which may prevent the destructor from run after the object is not used any longer in your application code.

The explicit call of `Unregister` will release any references to the `APMModule` also in the Java native library (JNI).

Syntax

```
public void Unregister();
```

4.6.3 Method RegisterEvent (Java)

It is required and important to register the events used in your application in order to get the correct event names and parameter descriptions as provided. The `RegisterEvent`-method will tell the app.telemetry server a name for every event-ID and optionally parameter descriptions for the parameters used in the event calls.

Syntax

```
public void RegisterEvent(long eventid, String eventDescription, String
parameterDescription);
```

Parameters:

- `eventid (long)`: Unique ID of event which is used later on for every instrumentation point.
- `eventDescription (String)`: Description string of event. A describing name for that event.

- `parameterDescription (String)`: Name of event parameters. If more parameters are used in one event, separate the parameter names by semi-colon (`;`). For example: "`name;count;size`"

Return Value:

This function does not return any value.

Usage Example

```
public class InstrumentedApplication {
  public static final int EVENT_HTTP_GET = 200;
  public static final int EVENT_HTTP_POST = 201;
  public static final int EVENT_REQUEST = 210;
  public static final int EVENT_PARAM_IN = 211;
  public static final int EVENT_PARAM_OUT = 212;
  public static final int EVENT_PARAMS = 213;

  public void registerEvents(APMModule module) {
    module.RegisterEvent(EVENT_HTTP_GET, "HTTP Get Request", "");
    module.RegisterEvent(EVENT_HTTP_POST, "HTTP Post Request", "");
    module.RegisterEvent(EVENT_REQUEST, "Process HTTP Request", "");
    module.RegisterEvent(EVENT_PARAM_IN, "Process HTTP Request - Enter",
"request");
    module.RegisterEvent(EVENT_PARAM_OUT, "Process HTTP Request - Leave",
"response");
    module.RegisterEvent(EVENT_PARAMS, "Process HTTP Request - Evaluate
Parameters", "id;thread;stock");
  }

  public static void main(String[] args) {
     // create application and module
     APMApplication app = new APMApplication();
     APMModule mod = new APMModule();

     registerEvents(mod);     // register all events

     // request and event calls
     app.CreateContext("filtervalue");
     mod.Event(EVENT_PARAM_IN, APM.EVENT_LEVEL_LOG, APM.FLAG_ENTER,
"request-data...");
     //...
     mod.Event(EVENT_PARAM_OUT, APM.EVENT_LEVEL_LOG, APM.FLAG_LEAVE,
"response-data...");
     app.ReleaseContext();
  }
}
```

4.6.4 Method Event (Java)

Call the simple `Event`-method to log events without parameters.

Only the event-ID and the flags define the representation of the event that will be displayed with the registered event name for the used id.

This method is part of the APMModule class and has to be called on your own module implementation class.

Syntax

```
public void Event(long eventid, byte level, bytes flags);
```

Parameters:

- `eventid (long)`: id of the event being processed. Should be defined as constant in your module implementation class.
- `level (byte)`: event detail level for filtering. For the list of available log levels see chapter 4.2.2 “*Log Level (Java)*”.
- `flags (byte)`: flags to mark events with Enter-, Leave- and/or Wait-semantic. More than one flags can be added together with a logic OR (`|`). For the list of available flags see chapter 4.2.4 “*Predefined Events (Java)*”.

Return Value:

This function does not return any value.

Usage Example

```
private void processData(String input) {
    // ...
    // fire enter event without parameters ...
    m_module.Event(EVENT_REQUEST, APM.EVENT_LEVEL_NORMAL,
APM.FLAG_ENTER);

    // fire event with String parameter ...
    m_module.EventStr(EVENT_PARAM_IN, APM.EVENT_LEVEL_DETAIL,
APM.FLAG_NONE, input);

    // ... process the data and do all the internal work ...

    // fire event with generic parameters ...
    APMParameter param = new APMParameter();
    param.AddInt64(id);
    param.AddInt32(threads);
    param.AddInt32(stock);
    m_module.EventParam(EVENT_PARAMS, APM.EVENT_LEVEL_DEBUG,
APM.FLAG_NONE, param);

    // ...
    // fire leave enter event without parameters ...
    m_module.Event(EVENT_REQUEST, APM.EVENT_LEVEL_NORMAL,
APM.FLAG_LEAVE);
}
```

4.6.5 Method EventStr (Java)

Call the `EventStr`-method to log events with a single textual (string) parameter.

In addition to the event-ID (with the registered event name) and the flags a single string parameter value will be logged.

For more information see 4.6.4 “*Method Event (Java)*”.

Syntax

```
public void EventStr(long eventid, byte level, bytes flags, String
param);
```

Parameters:

- `eventid (long)`: id of the event being processed. Should be defined as constant in your module implementation class.
- `level (byte)`: event detail level for filtering. For the list of available log levels see chapter 4.2.2 "*Log Level (Java)*".
- `flags (byte)`: flags to mark events with Enter-, Leave- and/or Wait-semantic. More than one flags can be added together with a logic OR (`|`). For the list of available flags see chapter 4.2.4 "*Predefined Events (Java)*".
- `param (String)`: a string parameter to be logged – can be any textual or preformatted message.

Return Value:

This function does not return any value.

Usage Example

`See` 4.6.4 "*Method Event (Java)*"

4.6.6 Method EventParam (Java)

Call the `EventParam`-method to log events with a generic parameter structure - with any number and combination of textual (string) values and integer values (32- and 64-bit).

To use the `EventParam`-method you have to allocate a new `APMParameter`. With a few helper methods you can add values to this parameter object and finally you can use the parameter object in the event function to be logged. For details how to allocate and fill the `APMParameter` object see chapter 4.7 "*APMParameter Class and Methods (Java)*".

Syntax

```
public void EventParam(long eventid, byte level, byte flags, APMParameter param);
```

Parameters:

- `eventid (long)`: id of the event being processed. Should be defined as constant in your module implementation class.
- `level (byte)`: event detail level for filtering. For the list of available log levels see chapter 4.2.2 "*Log Level (Java)*".
- `flags (byte)`: flags to mark events with Enter-, Leave- and/or Wait-semantic. More than one flags can be added together with a logic OR (`|`). For the list of available flags see chapter 4.2.4 "*Predefined Events (Java)*".
- `param (APMParameter)`: the parameter object containing any number of values to be logged.

Return Value:

This function does not return any value.

Usage Example

`See` 4.6.4 "*Method Event (Java)*"

4.7 APMParameter Class and Methods (Java)

The `APMParameter`-class is an object structure for storing and managing parameter values that can be used for logging complex value structures with more than one or different type of values.

See also 4.6.6 “*Method EventParam (Java)*”.

4.7.1 Constructor APMParameter (Java)

Call the `APMParameter`-constructor to allocate a new parameter object where you can add any number of textual (string) values and integer values (32- and 64-bit).

Syntax

```
public APMParameter();
```

4.7.2 Method AddInt32 (Java)

Call the method `AddInt32` on your `APMParameter`-instance to add a simple integer value (32-bit) to that object.

Syntax

```
public void AddInt32(int value);
```

Parameters:

- `value (int)`: 32-bit integer value to be added to the parameter object.

Return Value:

This function does not return any value.

Usage Example

`See` 4.6.6 “*Method EventParam (Java)*”.

4.7.3 Method AddInt64 (Java)

Call the method `AddInt64` on your `APMParameter`-instance to add a simple integer value (64-bit) to that object.

Syntax

```
public void AddInt64(long value);
```

Parameters:

- `value (long)`: 64-bit integer/long value to be added to the parameter object.

Return Value:

This function does not return any value.

Usage Example

`See` 4.6.6 “*Method EventParam (Java)*”.

4.7.4 Method AddString (Java)

Call the method `AddString` on your `APMParameter`-instance to add a textual (string) value to that object.

Syntax

```
public void AddString(String value);
```

Parameters:

- `value (String)`: textual parameter value to be added to the parameter object.

Return Value:

This function does not return any value.

Usage Example

`See` 4.6.6 “*Method EventParam (Java)*”.

5 JavaScript API Reference

The Fabasoft app.telemetry JavaScript API was introduced in the Summer Release 2009 and allows you to instrument any JavaScript application with instrumentation points to see what's going on in complex web applications.

In order to use the JavaScript API in your web application, you only have to include the softwaretelemetry.js SDK file and use the predefined functions and constants.

5.1 Constants and Configuration (JavaScript)

All of the constants mentioned below are defined in the JavaScript API file `softwaretelemetry.js` on the global object called "`apm`".

5.1.1 Version (JavaScript)

`apm.version` contains the version number of the used script.

Example: `alert(apm.version) // => "17.2.7"`

5.1.2 Flags (JavaScript)

Flags are used to specify the type of an event and are passed as argument to every event call.

Name	Value	Description
`apm.FLAG_NONE`	`0`	normal event
`apm.FLAG_ENTER`	`1`	indicates a starting point
`apm.FLAG_LEAVE`	`2`	indicates an ending point
`apm.FLAG_WAIT`	`4`	indicates a wait condition
`apm.FLAG_WARNING`	`8`	mark event as warning
`apm.FLAG_ERROR`	`16`	mark event as error
`apm.FLAG_PROCESS_INFO`	`32`	force writing *SessionInfo* just before the event

5.1.3 Log Level (JavaScript)

An event will only be logged if the event-level is lower or equal to the value selected in the session or log-definition.

Name	Value	Description
`apm.EVENT_LEVEL_LOG`	`0`	for events that should always be included and for logging parameters for the request overview
`apm.EVENT_LEVEL_NORMAL`	`50`	for general events for normal recording sessions

apm.EVENT_LEVEL_DETAIL	60	for detail level events for detailed session analysis
apm.EVENT_LEVEL_DEBUG	70	event with debugging information - highest level of precision with huge data amount and performance impact

5.1.4 Predefined Events (JavaScript)

The SDK already provides a set of predefined events used for general purpose. For a full listing of all available predefined events see the generic chapter 1.6 Predefined Events.

All predefined events are declared and available as global constants on the base variable "apm".

Error/Trace Events:

Use these standard events to mark error, warning or info conditions in your application:

- EVENT_ERROR
- EVENT_WARNING
- EVENT_INFO
- EVENT_TRACE

5.1.5 Predefined Application Value Keys and Names (JavaScript)

View the language agnostic Application Values section for an explanation of the properties listed below.

All predefined application value keys and names are available as public constants on the global apm object.

- VALUE_VERSION_KEY with name VALUE_VERSION_NAME

5.1.6 Environment Configuration (JavaScript)

Some global settings for the JavaScript SDK are configured globally on the environment object "apm.env":

Name	Default Value	Description
apm.env.url	"web.telemetry"	The address to which the JavaScript API posts its data. **Important:** this URL cannot point to a different domain or host as browser security prohibits this! You can of course map a virtual directory to a different server using a load-balancer or similar methods.
apm.env.active	true	With this Boolean value you can switch the JavaScript API *on* or *off*
apm.env.flushtimeout	5000 (= 5 sec)	Delay for first telemetry data cache flush. You can read the currently used flushtimeout (in ms) here, but you need to

		use `apm.SetTimeout` to change the timeout … see 5.2.1 “*Method SetTimeout (JavaScript)*”
`apm.env.flushinterval`	`60000` (= 1 min)	Telemetry data cache flush interval (in ms). You can read the currently used `flushinterval` (in ms) here, but you need to use `apm.SetInterval` to change the interval. … see 5.2.2 “*Method SetInterval (JavaScript)*”
`apm.env.eventsperflush`	`500`	Number of telemetry events sent with each cache flush.
`apm.env .reregistermodulestimeout`	`30000`	Resync module registration on changes after this timeout (ms).
`apm.env.reregistertimeout`	`600000` (= 10 min)	Interval (in ms) when synchronizing full registration information with agent/server.
`apm.env.exceptioncallback`	`null`	This callback is called (if set) when an exception occurred. For details see 5.1.6.1 “*Exception Callback (JavaScript)*”
`apm.env.navigationtiming`	`true`	Insert Navigation Timing data before the first call to CreateContext to add visibility into the loading process of the browser to start your application. You may want to disable this if your application expects to start up after the page finished loading.

5.1.6.1 Exception Callback (JavaScript)

This callback is called (if set) when an exception occurred.

There are a number of cases when this can occur:

- Data could not be transmitted to the Fabasoft app.telemetry WebAPI service.
- Some other error happened when trying to send data to the Fabasoft app.telemetry WebAPI service.
- The user callback has thrown an exception.

Syntax

```
apm.env.exceptioncallback = function(exobj, msg);
```

Parameters:

- `exobj (object)`: the JavaScript exception object or `null` if not available.
- `msg (String)`: String with a textual description of the occurred error.

Return Value:

The return value is ignored.

Remarks:

If `apm.env.exceptioncallback` is not set (`null`), the error message and a back trace will be logged to the JavaScript (Firebug)-Console (if available).

Usage Example

```
apm.env.exceptioncallback = function(exobj, msg) {
    alert("Exception:\nObject: " + exobj + "\nMsg: " + msg);
}
```

5.2 Configuration Functions (JavaScript)

Besides setting the basic configuration variables (like `apm.env.url`) you can modify some parameters for your needs with the following setup functions described in the next sub chapters.

5.2.1 Method SetTimeout (JavaScript)

Call `apm.SetTimeout` to change the initial timeout (delay) after which new events are being sent to the Fabasoft app.telemetry WebAPI service.

Syntax

```
apm.setTimeout(timeout);
```

Parameters:

- `timeout`: Timeout/delay in milliseconds after which new events are sent to the Fabasoft app.telemetry WebAPI service.

Return Value:

This function does not return any value.

Remarks:

You can read the currently used interval from `apm.env.flushtimeout`. Altering this value directly is not recommended.

Usage Example

```
// if you know that an action will take 2 seconds you might raise the
// timeout to ensure that all events are done and save some requests
// to the server.
apm.SetTimeout(3000);
```

5.2.2 Method SetInterval (JavaScript)

Call `apm.SetInterval` to change the interval in which the JavaScript API talks to the Fabasoft app.telemetry WebAPI service.

Syntax

```
apm.setInterval(interval);
```

Parameters:

- `interval`: Interval in milliseconds in which the Fabasoft app.telemetry WebAPI service is called to refresh registration and session information.

Return Value:

This function does not return any value.

Remarks:

- You can read the currently used interval from `apm.env.flushinterval`. Altering this value directly has no effect.
- Using a very small interval is a bad idea because some browsers limit the count of concurrent connections opened with `XMLHttpRequest`.

Usage Example

```
apm.SetInterval(3000);
```

5.2.3 Method Flush (JavaScript)

Call `apm.Flush` to trigger immediately sending events to the Fabasoft app.telemetry WebAPI service.

Syntax

```
apm.Flush(sendAllEvents);
```

Parameters:

- `sendAllEvents`: If `true` all available events are transmitted to the server asynchronously. If `false` a set of events will be transmitted in an asynchronous manner.

Return Value:

This function does not return any value.

5.3 Registration Functions (JavaScript)

5.3.1 Method RegisterApplication (JavaScript)

Call this function once to provide information about the context of the application.

Syntax

```
apm.RegisterApplication(appname, appid, apptiername, apptierid);
```

Parameters:

- `appname`: The name of your application
- `appid`: The Id of your application
- `apptier`: The tier inside the application
- `apptierid`: Id to distinguish multiple services of one application tier

Return Value:

This function does not return any value.

Remarks:

- Your application must be registered before any events can be fired.
- The communication with the Fabasoft app.telemetry WebAPI service is setup when fist calling `apm.RegisterApplication`.

Usage Example

```
apm.RegisterApplication("Software-Telemetry", "Demo", "Browser",
"browser");
```

5.3.2 Method RegisterApplicationValue (JavaScript)

Call RegisterApplicationValue to register additional static information about your application such as the version number.

Syntax

```
apm.RegisterApplicationValue(valueKey, valueName, value);
```

Parameters:

- `valuekey`: The application defined key of the value or the predefined VALUE_VERSION_KEY. Limited to 256 utf-8 encoded Bytes. Keys starting with “apm:” are res erved for internal use.
- `valueName`: The display name of the property (VALUE_VERSION_NAME, for the example above). Limited to 512 utf-8 encoded Bytes.
- `value`: The value of the property. Limited to 32768 utf-8 encoded Bytes.

Return Value:

This function does not return any value.

5.3.3 Method RegisterModule (JavaScript)

Call this function to register modules inside your application.

Syntax

```
apm.RegisterModule(modulename);
```

Parameters:

- `modulename` Desired name of your module.

Return Value:

- This function returns an object which you need to fire and register events.

Remarks:

- Different modules are shown as different bars during the analysis.
- For generic instrumentation of XMLHttpRequests there already exists a predefined module which can be obtained by calling *apm.GetXMLHTTPModule()*. (see 1.5.2 “*XMLHttpRequest*”)

Usage Example

```
var page = apm.RegisterModule("Page");

page.RegisterEvent(1, "ButtonClicked", "ArticleID");
page.RegisterEvent(2, "PageLoaded", "event");

var module2 = apm.RegisterModule("AJAX Status");
module2.RegisterEvent(3, "CheckServices", "");
module2.RegisterEvent(4, "ServiceStatus", "status");

var xhrMod = apm.GetXMLHTTPModule();
```

5.3.4 Method RegisterEvent (JavaScript)

Call `RegisterEvent` on a registered module instance object to provide information about every `eventid` used for event tracing.

Syntax

```
<module-instance>.RegisterEvent(id, name, param);
```

Parameters:

- `eventid`: This is the id of the event you want to register.
- `description`: The description of the event.
- `parameterdescription`: Name of event parameters. If more parameters are used in one event, separate the parameter names by semi-colon (`;`). For example: "`name;count;size`"

Return Value:

This function does not return any value.

Remarks:

If an event has more than one parameter you need to separate the `parameterdescription` with semicolons (`;`).

Usage Example

```
var module = apm.RegisterModule("AJAX Request");

module.RegisterEvent(4, "RequestResult", "status;status Text");
```

5.3.5 Method RegisterFilterValue (JavaScript)

Call `apm.RegisterFilterValue` for each valid filter value.

Syntax

```
apm.RegisterFilterValue(value, description);
```

Parameters:

- `value`: *value* is the raw filter value.
- `description`: This is the description which is presented in the Client-User-Interface as filter.

Return Value:

This function does not return any value.

Usage Example

```
apm.RegisterFilterValue("Article ID", "Description for `Article ID` filter.");
```

5.4 Context Handling Functions (JavaScript)

5.4.1 Method CreateContext (JavaScript)

Call `APMCreateContext` to create a new Software-Telemetry context. If the filter value matches a currently started session filter, logging is started for the current execution flow. Make sure to call `APMReleaseContext` when exiting the context.

Syntax

```
apm.CreateContext(contextid, filter);
```

Parameters:

- `contextid`: The current context or `null`. You may call `CreateContext` multiple times to provide additional filter parameters. To do this you have to provide the previously created context id as the `contextid` input parameter in subsequent calls to `CreateContext` so that the `CreateContext` can be associated with the same request. To start a new context pass `null` as the `contextid` to get a new *contextid* as a return value from the `CreateContext` function.
- `filter`: This value is used to match a filter of a started Software-Telemetry session. You should register filter values used by calling `RegisterFilterValue` (see 5.3.5 "*Method RegisterFilterValue (JavaScript)*")

Return Value:

- If the API is active this function returns a new context token, otherwise `null` will be returned.

Remarks:

- See chapter 1.7.3 "*Application Properties*" for more information on the app.telemetry request concept.
- See 5.4.2 "*Method ReleaseContext (JavaScript)*"

Usage Example

```
var contextid = apm.CreateContext(null, "filter1");
// ... proceed with application logic and fire events
apm.ReleaseContext(contextid);
```

5.4.2 Method ReleaseContext (JavaScript)

Call `APMReleaseContext` to finish logging this request. Make sure to call `apm.ReleaseContext` for every request created with apm.CreateContext.

Syntax

```
apm.ReleaseContext(contextid);
```

Parameters:

- `contextid`: The current context obtained by the corresponding `CreateContext` call.

Return Value:

This function does not return any value.

Remarks:

- If you don't call `APMReleaseContext` the request keeps the *running* status until the request-timeout has been reached.
- See 5.4.1 “*Method CreateContext (JavaScript)*”

Usage Example

```
var contextid = apm.CreateContext(null, "filter");
// fire events,...
apm.ReleaseContext(contextid);
```

5.4.3 Method GetContext (JavaScript)

Call `apm.GetContext` to get a context handle, which can be passed to another thread or process. This context information is used by the analysis components to track the control flow through the distributed environment.

The acquired context handle has to be submitted (either by transmitting over the network or in another way) to the thread/process being called which associates the calling thread with itself by passing the context to the `apm.AttachContext`-method.

Syntax

```
apm.GetContext(contextid);
```

Parameters:

- `contextid`: the current context (obtained by CreateContext).

Return Value:

- If the API is active this function returns a new *context-token*, otherwise `null` will be returned.

Remarks:

- See 1.8.1 “*Passing Context*”
- See 5.4.1 “*Method CreateContext (JavaScript)*”, 5.4.4 “*Method AttachContext (JavaScript)*”

Usage Example

```
var contextid = apm.CreateContext(null, "AJAX");
var contextToken = apm.GetContext(contextid);

// pass contextToken to another function or another host
//  (for example to the webserver via an x-apm-telemetry-context HTTP
```

```
header).
//another function can use the token to asynchronously do some work
//  (for example loading additional data with AJAX)
apm.ReleaseContext(contextid);
```

5.4.4 Method AttachContext (JavaScript)

Call `apm.AttachContext` to restore the context acquired through `apm.GetContext` in another thread, process or execution flow.

Syntax

```
apm.AttachContext(contextid, token);
```

Parameters:

- `contextid`: The current context or `null`. If you attach to a context within a previously created context using `AttachContext`, pass the existing contextid as an input parameter. To start a new context pass `null` as the contextid to get a new contextid as a return value from the `AttachContext` function.
- `token`: The context token which you want to attach to. You get such a context token by calling `apm.GetContext`.

Return Value:

- If the API is active this function returns a new *context-id* if the `contextid` parameter was `null`, otherwise `contextid` will be returned.

Remarks:

- See 1.8.1 “*Passing Context*”,
- See 5.4.3 “*Method GetContext (JavaScript)*”

Usage Example

```
var contextid = apm.AttachContext(null, token);
//proceed with application logic and fire events
apm.ReleaseContext(contextid);
```

5.4.5 Method GetSyncMark (JavaScript)

Call `apm.GetSyncMark` to get a synchronization handle that can be passed to another thread or process to synchronize control flow.

Syntax

```
apm.GetSyncMark(contextid);
```

Parameters:

- `contextid`: This is the context for which you want to get a sync-mark.

Return Value:

- The return value is a *Base64-encoded string* which represents the *token*, or `null` if the API is inactive.

Remarks:

- See 1.8.2 "*Synchronizing Timeline with SyncMarks*
- See 5.4.6 "*Method SetSyncMark (JavaScript)*"

Usage Example

```
var contextid = apm.CreateContext(null, "Test");
var token = apm.GetSyncMark(contextid);
```

5.4.6 Method SetSyncMark (JavaScript)

Call `apm.SetSyncMark` to pass a synchronization handle you got from `apm.GetSyncMark` for synchronizing the sequence of events.

Syntax

```
apm.SetSyncMark(contextid, token);
```

Parameters:

- `contextid`: This is the context for which you want to set a sync-mark.
- `token`: The sync-mark token you got from `GetSyncMark` form another thread/process.

Return Value:

This function does not return any value.

Remarks:

- See 1.8.2 "*Synchronizing Timeline with SyncMarks*
- See 5.4.5 "*Method GetSyncMark (JavaScript)*"

Usage Example

```
// registration,...
var context = apm.CreateContext(null, "Test");
// set the sync-mark we got from the web service (if any)
var token = document.getElementById('apmcontext').value; //e.g.: passed
in an hidden field
// or an use the response header of an AJAX call
token = xhr.getResponseHeader("x-apm-telemetry-syncmark");
if (token) {
    apm.SetSyncMark(context, token);
}
module.Event(context, g_idLoaded, apm.LEVEL_DETAIL, apm.FLAG_NONE, "page
loaded");
//...
apm.ReleaseContext(context);
```

5.5 Event Functions (JavaScript)

5.5.1 Method Event (JavaScript)

Call `Event` to log events with any number of parameters.

Syntax

```
<module-instance>.Event(contextid, id, level, flags, [params...]);
```

Parameters:

- `contextid`: This is the context in which the event happens and is to be logged.
- `id`: id of the event being processed (should have been registered with `RegisterEvent` – see 5.3.4 "*Method RegisterEvent (JavaScript)*").
- `level`: event detail level for filtering. For the list of available log levels see chapter 5.1.3 "*Log Level (JavaScript)*".
- `flags`: flags to mark events with Enter-, Leave- and/or Wait-semantic. More than one flags can be added together with a plus (`+`). For the list of available flags see chapter 5.1.2 "*Flags (JavaScript)*".
- `params...`: You can specify any number of parameters after the `flags` parameter. `null` parameters are also allowed - even if they are in between normal parameters (`null` parameters are replaced by empty string parameters if needed).
  Note: do not specify objects as parameters!

Return Value:

This function does not return any value.

Remarks:

- See 5.3.4 "*Method RegisterEvent (JavaScript)*", 5.1.4 "*Predefined Events (JavaScript)*"

Usage Example

```
var module = apm.RegisterModule("AJAX Request");
var context = apm.CreateContext(null, "filter1");
module.Event(context, 7, apm.LEVEL_DEBUG, apm.FLAG_NONE, "param1", null,
"param3", 4);
apm.ReleaseContext(context);
```

5.5.2 Method SendXMLHttpRequest (JavaScript)

Call `apm.SendXMLHttpRequest` to do a fully instrumented *XMLHttpRequest* including passing `x-apm-telemetry-context` headers to the webserver and handling `x-apm-telemetry-syncmark` response headers for End-2-End instrumentation.

Syntax

```
apm.SendXMLHttpRequest(request, async, value, callback, context);
```

Parameters:

- `request`: The *XMLHttpRequest* object.
- `async`: Set this to `true` if you want to do an asynchronous request.
- `value`: This is the data sent with the request.
- `callback`: This callback is executed on every `readystatechange` (it does not apply for synchronous requests).
- `context`: This is your current context in which the request is being sent.

Return Value:

This function does not return any value.

Usage Example

```
apmCheckServices = function() {
    var request = new XMLHttpRequest();
    var context = apm.CreateContext(null, "AJAX Request");
    var token = apm.GetContext(context);
    module.Event(context, 3, apm.LEVEL_NORMAL, apm.FLAG_ENTER);
    request.open("GET", "./AJAXHandler.ashx", true);
    apm.SendXMLHttpRequest(request, true, null, function(xmlhttp,
context) {
        if (xmlhttp.readyState == 4) {
            try {
                var json_obj = JSON.parse(xmlhttp.responseText);
                module.Event(context, 4, apm.LEVEL_NORMAL, apm.FLAG_NONE,
"stockService: " + json_obj.stock);
                updateStatus('stockService', json_obj.stock);
                module.Event(context, 4, apm.LEVEL_NORMAL, apm.FLAG_NONE,
"repositoryService: " + json_obj.repository);
                updateStatus('repositoryService', json_obj.repository);
            } catch (e) {
                module.Event(context, 4, apm.LEVEL_NORMAL, apm.FLAG_NONE,
"Exeption(" + e.name + "): " + e.description);
            }
        }
    }, context);
    module.Event(context, 3, apm.LEVEL_NORMAL, apm.FLAG_LEAVE);
    apm.ReleaseContext(context);
}
```

5.6 Fabasoft app.telemetry Button (JavaScript)

5.6.1 Method Report (JavaScript)

Call `apm.Report` to create a new feedback report.

The filter value will be matched to the description using the registered filter values. Use the textual description of the feedback entered by the user or generated by the application to identify the report session later again.

Syntax

```
apm.Report(contextid, filter, reportkey, description);
```

Parameters:

- `contextid`: This is the context in which the report is generated.
- `filter`: The raw filter value.
- `reportkey`: A key value must be generated by the calling application and can be used to associate further descriptions to a feedback.
- `description`: Application generated or entered by the user. The maximum length of the description is **32768** Bytes. (Prior to Fabasoft app.telemetry 2010 Fall Release (5.3) this parameter was limited to 256 Bytes.)

Return Value:

This function does not return any value.

Usage Example

```
var module = apm.RegisterModule("User Reaction");
var context = apm.CreateContext(null, "filter1");
module.Event(context, 7, apm.LEVEL_DEBUG, apm.FLAG_ENTER+apm.FLAG_WAIT);
alert("Press Ok to continue");
module.Event(context, 7, apm.LEVEL_DEBUG, apm.FLAG_LEAVE+apm.FLAG_WAIT);
var repkey = "myreport_" + Math.random();
apm.Report(context, "filter1", repkey, "Application has been waiting for
user input");
apm.ReportValue(context, repkey, "username", "John Doe");
apm.ReleaseContext(context);
```

5.6.2 Method ReportValue (JavaScript)

Call `apm.ReportValue` to add a key/value pair to an existing report.

Syntax

```
apm.ReportValue(contextid, reportkey, key, value);
```

Parameters:

- `contextid`: This is the context in which the report is generated.
- `reportkey`: match the reportkey of the `Report` function to associate the value with the report session.
- `key`: A key describes the meaning of the value. Only one value can be associated to a key in one report session.
- `value`: Textual value (max. 32 kBytes).

Return Value:

This function does not return any value.

Usage Example

`See` 5.6.1 „*Method Report (JavaScript)*“

5.6.3 Method ReportContent (JavaScript)

Call `apm.ReportContent` to add content to an existing report.

Syntax

```
apm.ReportContent(reportkey, filename, value, encoding, mimetype);
```

Parameters:

- `reportkey`: match the reportkey of the `Report` function to associate the content with the report session.
- `filename`: A filename describing the content of the file. (max. 256 Bytes) – this is the name of the file in the reported session with which the data is persisted.
- `value`: This is the content – a string or byte-array to attach to the report.
- `encoding` (optional): The encoding type of the value content. Supported encoding types are:
  - "`plaintext`" (=default: if missing or `null`): string value will be written directly into the target file.
  - "`base64`": the passed content `value` must be *Base64-encoded* and will be decoded as *Base64-value* on the server for writing into the target file. This encoding allows you to attach binary data, images or similar to the report session.
- `mimeType` (optional): The MIME-type helps to identify the uploaded file content. You can provide any valid mime type string:
  - "`text/plain`" (=default: if missing or `null`)
  - "`image/png`"
  - "`application/octet-stream`"
  - etc.

Return Value:

This function does not return any value.

Usage Example

```
var module = apm.RegisterModule("User Reaction");
var context = apm.CreateContext(null, "filter1");
module.Event(context, 7, apm.LEVEL_DEBUG, apm.FLAG_ENTER+apm.FLAG_WAIT);
alert("Press Ok to continue");
module.Event(context, 7, apm.LEVEL_DEBUG, apm.FLAG_LEAVE+apm.FLAG_WAIT);
var reportkey = apm.utils.Encode64(apm.utils.CreateContextToken());

apm.Report(context, "filter1", reportkey, "Application has been waiting
for user input");
apm.ReportValue(context, reportkey, "username", "John Doe");
apm.ReportContent(reportkey, "page.html",
document.documentElement.outerHTML);
apm.ReportContent(reportkey, "screenshot.png", base64encodedImageContent,
"base64", "image/png");
apm.ReleaseContext(context);
```

Remarks:

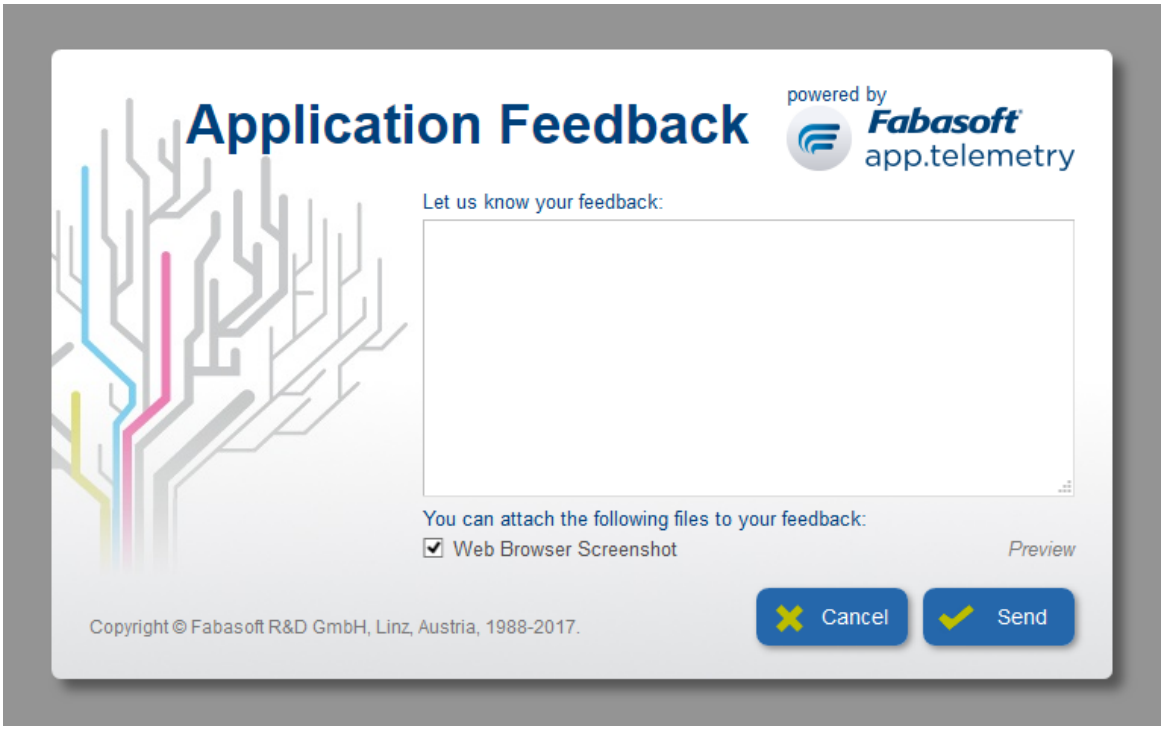
To allow applications to post random content, Fabasoft app.telemetry WebAPI supports posting content directly to the Fabasoft app.telemetry Web-API. Provide the `reportkey` and the `filename` as URL-parameters after the "`?ReportContent`" tag and the send the `value` as POST-data content.

Usage Example – ReportContent POST

```
var request = new XMLHttpRequest();
var url = apm.env.url + "?ReportContent";
url += "&reportkey=" + encodeURIComponent(reportkey);
url += "&filename=" + encodeURIComponent(filename);
request.open("POST", url, true);
request.send(value);
```

5.6.4 Method ReportDialog (JavaScript)

The JavaScript API provides a full-featured implementation of a report dialog to ask the user for feedback which is sent with the other report data. When used with modern web browsers also a web-based HTML5 browser screenshot is available.



5.6.4.1 Syntax and Usage

Call `apm.ReportDialog` to show the end user a feedback dialog which will invoke the `apm.Report` call with the description entered by the user and if defined it will append additional metadata information and files to the same report session.

Syntax

```
apm.ReportDialog(contextid, filter, reportkey, description, parentNode,
metadata, formOptions);
```

Parameters:

- `contextid` (optional): This is the context in which the report dialog is opened by the user (e.g. context of user button click). If you do not have an active context pass "`null`" and the API will create the required context internally. This context is released after the dialog is created and displayed!
- `filter` (required): The raw filter value.
- `reportkey` (required): A key value (string) must be generated by the calling application to associate further descriptions to a feedback (as unique value for every generated report). The key value can be used to attach further metadata/attachments.
- `description` (optional): Optionally a predefined description string may be passed to this function to provide a start-up text for the description input text field presented to the user within the feedback dialog – the user can change this text. If empty ("") the user will see an empty input text field in the dialog where he can input his own feedback text.
- `parentNode` (optional): This is the parent DOM-Node to which the created dialog is added, if you do not specify this parameter it defaults to `window.document.getElementsByTagName('BODY')[0]`.
- `metadata` (optional): The metadata object is an optional parameter to define additional metadata to display in the report dialog and attach to report session. The metadata object has the following format:

`_metadata` object Syntax/Example

```
function previewCallback(filename, fileObj) {
  // open preview window
}
function getContentCallback(filename, fileObj) {
  // return base64-encoded file content as string
}
var metadata = {
  fields: [    // custom editable fields (with label and input-control)
               // shown above feedback textbox
    {
      type: "text",             // = default
      name: "email",            // field name
      label: "E-Mail Address",  // field label (optional)
                                // ... if not set, name is used as label
      value: "test.user@mydomain.com",    // preset value
                                // (initially filled into the text box)
      validate: function(name, value) {   // validate function
        // that must return true on OK
        // or a text that is displayed otherwise (if invalid)
         if (value && value.length > 10) {
             return true;
          } else {
            return "Value is invalid - shorter than 10 chars!";
          }
      },
      html5type: "email",     // optional set the html5 input type
                              //   (for better handling on tablets)
      layout: "two-column"    // optional (default if not defined
                              //   is one-column with 2 rows)
    },
    {
      type: "checkbox",
      label: "My Checkbox",
      name: "check1",
```

```
      checked: true          // preselection
    },
    {
      type: "select",         // combo box
      label: "My Combo", name: "combo1",
      options: {
        "val_1": "Value 1",
        "val_2": "Value 2",
        "val_3": "Value 3"
      },
      value: "val_2"         // preselection
    },
    {
      type: "rating",       // radio-button rating control
      label: "Recommend", name: "rate2",
      count: 5,             // number of radio buttons for rating
      leftHint: "bad",
      rightHint: "good",  // optional hint text above radio buttons
      radioLabels: ["--","-","o","+","++"],  // radio button labels
                    // (below) can be customized (default: 0..count-1)
      value: 2,             // initial selection
      layout: "two-column"  // if not two-column, then a single-column
                            //   layout with 2 rows is used
    },
    {
      type: "property",
      label: "Readonly Property",
      name: "property1",
      value: "value_1"
    },
    { type: "hr" }   // Horizontal Line
  ],
  files: [    // custom files/contents to be uploaded with session
    {
      filename: "my-screenshot1.jpg",
      checked: true,       // default state of checkbox
      name: "screenshot",  // internal key to set id for checkbox HTML-
        // tag ("apm-chk-file-" + id) ... valid chars: [-A-Za-z0-9_]
      label: "Include Screenshot",  // label for list entry
      previewLabel: "Preview...",   // label for preview button
                                    // (if "" -> no preview button/link)
      previewCallback: previewCallback,  // callback to view file content
                                         //   (if null -> no button)
      getContent: getContentCallback,    // callback to get file content
(base64-encoded)
      mimeType: "image/jpeg"  // mime-type of file content
    },
    {
      filename: "my-system-information.txt",
      checked: true,
      name: "softwareinfo",
      label: "Include Software Repository/Registry",
      previewLabel: "Preview...",
      previewCallback: previewCallback,
      getContent: getContentCallback,
      mimeType: "text/plain"
    },
  ],
```

```
  infos: [        // list of key-value pairs for additional text
information
    {
      name: "user",         // name of attribute (sent to server)
      label: "Username",    // optional: display string for GUI
                            //   (if not set the name is displayed)
      value: "Test User"    // value of attribute (sent to server)
    },
    {name: "E-Mail", value: "test.user@mydomain.com"},
    {name: "version", label: "Client-Version", value: "1.2.3.4"}
  ]
};
apm.ReportDialog(null, filter, reportkey, description, parentNode,
metadata);
```

- `formOptions` (optional): Since version 2013 Fall Release this optional parameter can be used to pass additional parameters.
  - `formid`: to load a designed on-premise form via the WebAPI. This will prevent the default SDK dialog from being shown and will instead load the form with the given ID. Format: `formOptions = { formid: "<FORMnnnnn>" }`
  - `language`: to load a specific language translation of a designed on-premise form. If not defined the language of the HTML-page where the dialog is embedded will be used (`document.documentElement.lang`). If no form translation for the desired language can be found the default dialog without text translations will be loaded. Format: `formOptions = { language: "<xy>" }`

`_formOptions` object Syntax/Example

```
var formOptions = {
    formid: "FORM12345",
    language: "de",
};
apm.ReportDialog(null, filter, reportkey, description, parentNode,
metadata, formOptions);
```

Return Value:

This function does not return any value.

Remarks:

- Calling this dialog when the API is disconnected results in an alert with the message stored in `apm.env.report.reportingunavailable`, you can call `apm.IsConnected()` to check if this will happen.
- During the lifetime of the dialog all `iframe`, `object`, and `embed` tags are set to `visibility: hidden`, after closing the dialog all changed elements are set back to their original visibility value.

5.6.4.2 Customizing Report Dialog

The application developer can customize the texts used in the dialog and optionally can define the path where the image file resources are located.

To customize the dialog settings just overwrite the default values in the `apm.env.report` API variables:

Name (`apm.env.report. ...`)	Default Value	Description
`dialogtitle`	`"Application Feedback"`	Title displayed at the top of the dialog (heading)
`dialogpoweredby`	`"powered by"`	Powered by… text above logo
`poweredByLogo`	`<empty>`	optionally set a custom poweredBy logo – keep empty to use the default app.telemetry logo (will be set as `img-src` attribute)
`poweredByLogoLink`	`<empty>`	optionally define a custom link target for the powered-by logo (set as `a-href` attribute) – keep empty to use the default URL (http://www.apptelemetry.com)
`hidePoweredBy`	`false`	could be set to `true` to fully hide the powered-by logo and text
`hideBackgroundImage`	`false`	could be set to `true` to hide the background image tree to save some space on the left
`enableLogoLinkToHomepage`	`true`	if `true` the Logo represents a hyperlink to www.apptelemetry.com otherwise there will be no link but only the logo
`dialogtext`	`"Let us know your feedback:"`	Default label text for main feedback input box (shown above textarea).
`dialogtextminlength`	`0`	Minimum number of characters the user has to enter to be able to submit the feedback (if not fulfilled the `dialogtexttooshort`-text is displayed).
`dialogtextmaxlength`	`1000`	Maximum number of characters the user can enter in the text-field (`textarea` limited with `maxlength`-property) - if set to 0 the length is not limited (`maxlength`-property not set)
`dialogtexttooshort`	`"The feedback you entered is too short."`	Displayed error text when user tried to send the feedback but did not enter enough text (characters) into the textarea.
`dialogsubmittext`	`"Send"`	Label text for send/submit button.
`dialogcanceltooltip`	`"Cancel"`	Label text for cancel/close button.
`closeonescape`	`true`	If true the dialog will be closed (cancelled) on pressing the escape key.
`reportingunavailable`	`"The feedback functionality is currently`	Text shown when feedback functionality is not available (base URL not configured correctly or webAPI not reachable).

	`unavailable."`	
`imagepath (ignored)`	`null`	The imagepath parameter is not used any longer. Images (referenced from apmdialog.css) are loaded directly from the WebAPI
`scripting`	`true`	This setting allows you to disable any scripting field functions possibly used in additional form fields.

Additional Customization Variables (for providing metadata content):

In order to use the automatic *screenshot/softwareinfo* integration of the Fabasoft web browser plugin the following conditions must be fulfilled:

- Fabasoft web browser plugin is installed on every client machine using the feedback dialog
- The JavaScript part of this Fabasoft plugin (`fscplugin.js`) is included and loaded before the softwaretelemetry.js.
- Your server location is trusted for Fabasoft plugin usage in the registry of every client using the feedback dialog: `HKEY_CURRENT_USER\Software\Fabasoft\WebClient\TrustedSites` … with a „String"-key containing the application server URL as name and the value "1" as value. (E.g.: http://server.mydomain.com = 1)

Otherwise you can provide your own file attachments within the metadata files list including the callback functions for getting the content and previewing the file.

- `apm.env.report.screenshot = {enabled: true, label: 'Include Screenshot', previewlabel: 'Preview...'}` ... to customize the Plugin-based screenshot content attachment
- `apm.env.report.softwareinfo = {enabled: true, label: 'Include Software Repository/Registry', previewlabel: 'Preview...'}` ... to customize the Plugin-based screenshot content attachment
- `apm.env.report.displaymetadata = true`; ... set this option to false to hide additional info key-value pairs attached to the report session (information is only hidden but sent with report)
- `apm.env.report.metadatatitle = 'Your feedback will include the following information:'` ... section title label for additional info key-value pairs
- `apm.env.report.filestitle = 'You can attach the following files to your feedback:'` ... section title label for additional attached files (and screenshot/softwareinfo attachments)

Available event callbacks: (default they are not defined - `null`)

- `apm.env.report.onopen = function() { ... }`
  - The `onopen`-callback is invoked after the dialog has been created and inserted into the DOM, but before the `iframe`, `object` and `embed` elements are hidden.
- `apm.env.report.oncancel = function(data) { ... }`
  - The `oncancel`-callback is invoked when the user cancels the feedback dialog with the close/cancel button.
  - If the assigned callback function returns `false` the dialog will not be canceled/closed. In normal cases the function should return `true`.
  - The data argument passed to the `oncancel` callback is the `apm.env.report.data` variable which can be read - but should not be changed here - changes made to the

description property will be overwritten when the user presses OK(in case the `oncancel`-callback returns `false` and the user presses OK afterwards).

- `apm.env.report.onsubmit = function(data) { ... }`
  - The `onsubmit`-callback is invoked when the user presses the submit/OK button in the feedback dialog.
  - If the assigned callback function returns `false` the dialog will not be submitted (no report sent) and will stay open. In normal cases the function should return `true`. You can use this callback to validate the input and inform the user about what he missed to enter (min text-length, E-Mail address, whatever)
  - The data argument passed to the `onsubmit` callback is the `apm.env.report.data` variable which can be read and also modified, changes to the description property are not reflected in the input element the user sees and are overwritten the next time the user presses submit (if you returned false to keep the dialog open before).
- `apm.env.report.onclose = function() { ... }`
  - the `onclose`-callback is invoked after the dialog has been removed from the DOM, the `iframe`, `object` and `embed` elements are already set to their original visibility values.
- `apm.env.report.hideoverlay = true/false`
  - if set to `true` the dialog overlay is hidden (`display: none`) to allow custom overlay handling.
- The `apm.env.report.data` object:

Data Structure (`apm.env.report.data`)

```
apm.env.report.data = { contextid, filter, key, description, }
```

  - The `contextid`, `filter` and `key` values have already been assigned with the arguments passed by the initial call of apm.ReportDialog(...)
  - The `description` field contains the value the user entered in the text input field inside the dialog
  - e.g: the developer can use the `oncancel`/`onsubmit`-callback to get the value the user entered in the feedback dialog and use it in his own application logic.

Special parameter access:

- `env.report.dialognode`: this variable stores the DOM-node (DIV) used to display the feedback dialog.

Internal methods to control the dialog (should only be used in exceptional cases):

- `apm.utils.createReportDialog()` ... create and show dialog (is called by ReportDialog main function)
- `apm.utils.cancelReportDialog()` ... same as user cancels dialog (Close/X)
- `apm.utils.removeReportDialog()` ... remove dialog DIV (is called by cancelReportDialog and submitReportDialog)
- `apm.utils.submitReportDialog()` ... same as user presses submit/OK

5.6.4.3 Customizing Report Dialog Screenshot/Preview

Since Winter Release 2012 the app.telemetry feedback dialog supports an HTML5 screenshot to be taken and submitted via the feedback dialog without any browser plugin/addon. This feature is supported in all modern web browsers which support the HTML5-canvas element native.

To customize if the HTML5 screenshot should be enabled for your website and to customize the display texts used for the HTML5 screenshot attachment inside the feedback dialog use the following properties of the `apm.env.report` data object:

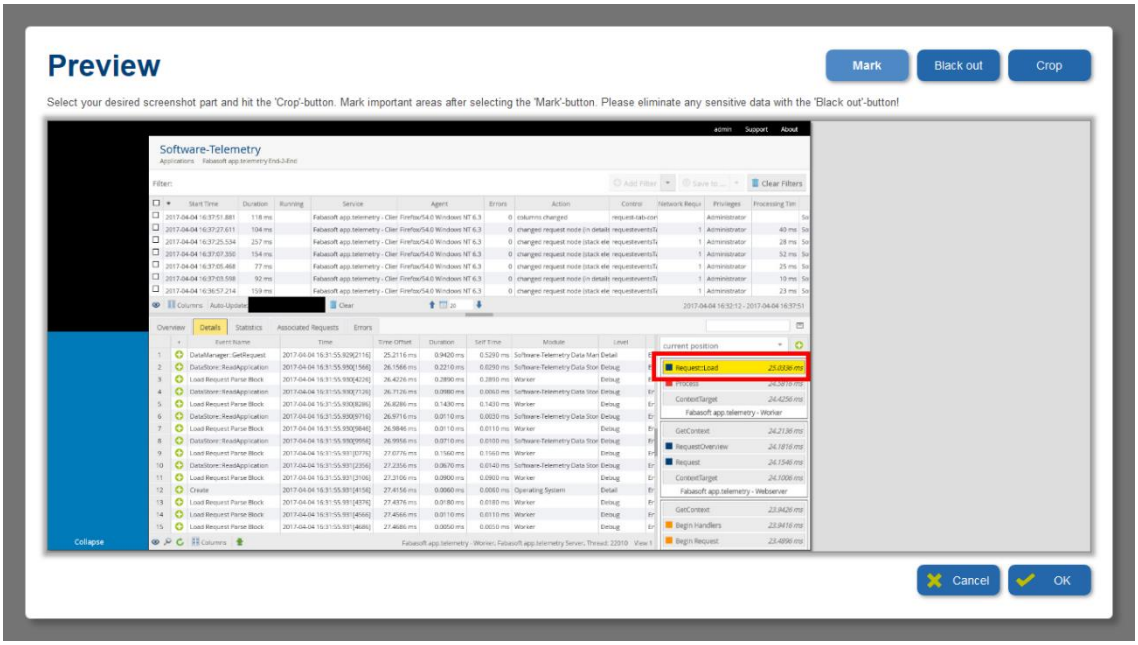
Name (`apm.env.report. ...`)	Default Value	Description
`enableHTML5Screenshot`	`true`	Enable (`true`) / disable (`false`) support and automatic rendering of HTML5 screenshot on opening the feedback dialog
`HTML5ScreenshotPreview`	`"Preview"`	display label for preview screenshot link
`HTML5ScreenshotLoading`	`"loading..."`	display label for preview screenshot link during loading/rendering of screenshot
`HTML5ScreenshotLabel`	`"Web Browser Screenshot"`	display label for HTML5 screenshot file attachment listing
`HTML5ScreenshotPreviewUnavailable`	`"N/A"`	display label for preview screenshot link if no screenshot is available or loading/rendering failed
`HTML5ScreenshotClosePreviewTitle`	`"Close"`	Button label for close button for preview window.
`HTML5ScreenshotLoadingTimeout`	`30000`	timeout (in msec) after which the loading/rendering of the HTML5 screenshot will be cancelled

Since version 2013 Summer Release the screenshot preview was redesigned and improved supporting the following new features:

- **Crop/cut** screenshot to desired size (especially to hide other application windows captured with the native Fabasoft plugin screenshot)
- **Mark** screenshot: mark interesting/important parts in the screenshot with a colored (red) rectangle
- **Redact**: hide parts you don't want to transmit with your feedback from the screenshot by painting those areas with black filled rectangles

These new features …

- require HTML5-support in the web browser (only modern browsers like Chrome, Firefox or IE9+ are supported)
- work for captured HTML5-screenshots of any web page and for native Fabasoft plugin screenshots



In order to customize the label and button texts of the new screenshot preview dialog use the following configuration parameters of the `apm.env.report` data object:

Name (`apm.env.report. ...`)	Default Value	Description
`screenshotPreviewTitle`	`"Preview"`	Displayed top title for the screenshot preview dialog.
`screenshotPreviewHelpText`	`"Select your desired screenshot part ..."`	This is the help text displayed in the preview dialog to explain the crop/mark features and give the user other hints for using this preview dialog.
`screenshotPreviewCropButtonLabel`	`"Crop"`	Button label for the “cut/crop” feature.
`screenshotPreviewMarkButtonLabel`	`"Mark"`	Button label for the “mark” feature.
`screenshotPreviewBlackoutButtonLabel`	`"Redact"`	Button label for the “redact” feature.
`screenshotPreviewCancelButtonLabel`	`"Cancel"`	Button label for the “cancel/close” preview dialog.
`screenshotPreviewOkButtonLabel`	`"OK"`	Button label for the “ok/apply” to accept modifications and close preview dialog.

5.6.4.4 Using Report Dialog Example

Usage Example

```
...
  <script type="text/javascript" src="softwaretelemetry.js"></script>
```

```
  <script language="javascript">
    function sendFeedbackFunction() {
      apm.env.report.dialogtitle = "Report Your Problems";
      apm.env.report.oncancel = function() {
        console.log("oncancel callback called");
        return true;
      };
      apm.env.report.onsubmit = function(data) {
        console.log("onsubmit callback called");
        console.log(["user sends feedback: ", data]);
        return true;
      };
      var metadata = {
        fields: [
          { name: "email", label: "E-Mail Address", value:
"user@mydomain.com",
            validate: function(name, value) {
              if (value && value.length > 10) { return true; }
              else { return "Value invalid - shorter than 10 chars!"; }
             }
          }
        ],
        files: [
          {
            filename: "test-screenshot.png",
            checked: true,
            name: "screenshot",
            label: "Application Screenshot",
            previewLabel: "Preview...",
            previewCallback: previewCallback,
            getContent: getImageContentCallback,
            mimeType: "image/png"
          },
          {
            filename: "test-system-information.txt",
            checked: false,
            name: "softwareinfo",
            label: "Application Software Info",
            previewLabel: "Long Preview Text...",
            previewCallback: previewCallback,
            getContent: getTextContentCallback,
            mimeType: "text/plain"
          }
        ],
        infos: [
          {name: "Username", value: "Test User (domain\\user.name)"},
          {name: "e-Mail", value: "test.user@mydomain.com"},
          {name: "Client-Version", value: "1.2.3.4"},
        ]
      };
      apm.ReportDialog(null, "myfilter", "myReportKey",
          "description passed via API call", null, metadata);
      return false;
    }
  </script>

  <!-- display a feedback button with the Fabasoft app.telemetry logo
         provided via stylesheet classes -->
```

```
<a href="#" title="Feedback" onclick="sendFeedbackFunction();">
  <span class="apm-feedback-icon apm-ic-logo20"></span>
</a>
```

5.7 Status Functions (JavaScript)

5.7.1 Method HasActiveContext (JavaScript)

Call `apm.HasActiveContext` to test, if a software-telemetry session is active for some logging level. This information should be used to avoid costly operations preparing parameters for events that are currently not logged.

Syntax

```
apm.HasActiveContext(contextid, level);
```

Parameters:

- `contextid`: This is the context for which you want to know if the event will be logged.
- `level`: The event level of the event you prepare parameters for.

Return Value:

- The return value is `true` if the event will be logged.

Usage Example

```
var context = apm.CreateContext(null, "filter value");
// application logic
if (apm.HasActiveContext(context, apm.LEVEL_DEBUG)) {
    // prepare parameters, set events,...
}
apm.ReleaseContext(context);
```

5.7.2 Method IsConnected (JavaScript)

Call `apm.IsConnected` to test, if the application has successfully registered to a Fabasoft app.telemetry agent/WebAPI. Your application must not make any assumptions based on the return value of this function as there are many reasons that may lead to a false being returned from this function even though the communication with the WebAPI be possible in general.

Syntax

```
apm.IsConnected();
```

Parameters:

This function does not take any arguments.

Return Value:

- This function returns true if the communication to the configured Web-API web service is successful.

Remarks:

- A call to this function does not result in an *XMLHTTPRequest* to the web service; instead, it returns the result of past communications, depending on when you call this function you may need to trigger a flush before.
- You can check if the API is active by checking `apm.env.active`. See 5.1.6 “*Environment Configuration (JavaScript)*” for details.
- Please note that versions prior to Fabasoft app.telemetry 2010 Fall Release (5.3) always returned false.

Usage Example

```
apm.RegisterApplication("Fabasoft app.telemetry", "", "Client", "");
apm.Flush(); // trigger a flush to make sure at least one request was
done
// (much) later:
if (apm.IsConnected()) {
    console.log("APM Software-Telemetry connected");
}
```

6 Android API Reference

Since version 2013 Winter Release the Fabasoft app.telemetry SDK was extended to also support mobile devices. In this case Android apps can be instrumented using the Fabasoft app.telemetry SDK for Android.

This SDK for Android is very similar to the basic Java SDK (see 4 "*Java API Reference*") with 2 exceptions:

1. The **data transport is HTTPS-based**. Instead of relying on a native library to communicate with the app.telemetry agent, the mobile SDK uses HTTPS-network calls to communicate with the Fabasoft app.telemetry WebAPI (which forwards the data to the Fabasoft app.telemetry Agent).
2. Most **API-methods require a thread-token** as an additional parameter.

Additionally the Android SDK provides a helper function to create a screenshot from the device screen that could be sent with your feedback: `APM.createScreenshot()`.

6.1 Prerequisites for Instrumenting Android Java Apps

In order to use the Fabasoft app.telemetry SDK for mobile devices (Android) you will need an installation of Fabasoft app.telemetry (Server, Agent + WebAPI) and ensure that the WebAPI web service is reachable from your mobile device by checking the WebAPI status page of your installation (`https://<your server>/web.telemetry`). This installation will receive the telemetry data and will be used to configure the infrastructure (agent, log pools, etc.).

Before starting to develop anything for your mobile device, you will need a development environment for Android and all device drivers to communicate with your device.

6.2 Configuration for Android Java Apps

The next required step is to include the Fabasoft app.telemetry SDK library for Android (`softwaretelemetry-android.aar`) as additional library into your project. This library is delivered with the app.telemetry installation media in the "`Developer/Android`" folder. The JavaDoc is next to the archive in a jar file (`softwaretelemetry-android-javadoc.jar`).

6.3 Instrumenting Android Java Apps

The technical background for instrumenting any Java-based application is already described in chapter 4 "*Java API Reference*".

In order to initialize the SDK for your mobile app you need to pass the Android app "*Context*" and the target URL of your app.telemetry WebAPI instance.

Syntax

```
static final String WEBAPI_URL = "http://<your server/web.telemetry";
IAPMDataTransport transport = new MyDataTransport()
APM.init(this.getApplicationContext(), WEBAPI_URL, transport);
APMApplication app = APM.RegisterApplication("Fabasoft app.telemetry",
"Test Application", "Android", "Test 1");
```

The differences about how to use this API for an Android-based Java application can be obtained from the sample project.

Usage Example (`APMDemoApplication.java`)

```
package com.example.feedback;

import android.net.Uri;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

import com.apptelemetry.apm.client.APM;
import com.apptelemetry.apm.client.APMModule;
import com.apptelemetry.apm.client.APMReportContent;
import com.apptelemetry.apm.client.IAPMDataTransport;
import com.apptelemetry.apm.client.IAPMDataTransportResponseCallback;

import android.view.View;
import android.widget.EditText;
import android.content.Intent;

public class MainActivity extends AppCompatActivity {
  APMModule moduleEvent;
  //change the following URL to YOUR app.telemetry server/WebAPI
  //it is the URL where the telemetry data and the feedback is sent to
  static final String WEBAPI_URL = "https://<server>/web.telemetry";

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    APM.init(this.getApplication(), WEBAPI_URL, new MyDataTransport());
    APM.RegisterApplication("Fabasoft app.telemetry", "Test Application",
"Android", "Test 1");
    moduleEvent = APM.getModule("Event");
    moduleEvent.RegisterEvent(1001, "sendFeedback", "message");
    moduleEvent.RegisterEvent(1002, "secondThread", "message");
  }
  public void sendFeedback(View view) {
    APMReportContent screenshot = APM.createScreenshot(view);
    long thread = APM.CreateContextTx(0, "sendFeedback");
    Intent intent = new Intent(this, FeedbackDialogActivity.class);
    EditText editText = (EditText) findViewById(R.id.edit_message);
    moduleEvent.EventStrTx(thread, 1001, APM.EVENT_LEVEL_NORMAL,
APM.FLAG_ENTER, editText.getText().toString());
    byte[] contextToken = APM.GetContextTx(thread);
    long thread2 = APM.AttachContextTx(0, contextToken);
    moduleEvent.EventStrTx(thread2, 1002, APM.EVENT_LEVEL_DETAIL,
APM.FLAG_ENTER, editText.getText().toString());
    moduleEvent.EventTx(thread2, 1002, APM.EVENT_LEVEL_NORMAL,
APM.FLAG_LEAVE);
    byte[] syncMark = APM.GetSyncMarkTx(thread2);
    APM.ReleaseContextTx(thread2);
    APM.SetSyncMarkTx(thread, syncMark);
    String message = editText.getText().toString();
    intent.putExtra(FeedbackDialogActivity.EXTRA_SCREENSHOT, screenshot);
    intent.putExtra(FeedbackDialogActivity.EXTRA_TOPIC, message);
    startActivity(intent);
    moduleEvent.EventTx(thread, 1001, APM.EVENT_LEVEL_NORMAL,
APM.FLAG_LEAVE);
    APM.ReleaseContextTx(thread);
  }
```

```
}

class MyDataTransport implements IAPMDataTransport {
  @Override
  public boolean isNetworkConnected() {
    return true;
  }
  @Override
  public boolean sendData(Uri uri, String contentType, byte[] bytes,
IAPMDataTransportResponseCallback callback) {
    // TODO: your implementation here
    MyResponseObject response = sendHttpPostRequest(uri, contentType,
bytes);
    callback.dataSent(response.getStatus(), response.getContentType(),
response.getResponseStream());
    return true;
  }
}
```

7 iOS API Reference

Since version 2013 Winter Release the Fabasoft app.telemetry SDK was extended to also support for mobile devices. In this case Apple iOS apps written in Objective-C can be instrumented using the Fabasoft app.telemetry SDK.

This SDK for iOS is technically very similar to the Android Java SDK (see 6 "*Android API Reference*") except for the programming language which is Objective-C.

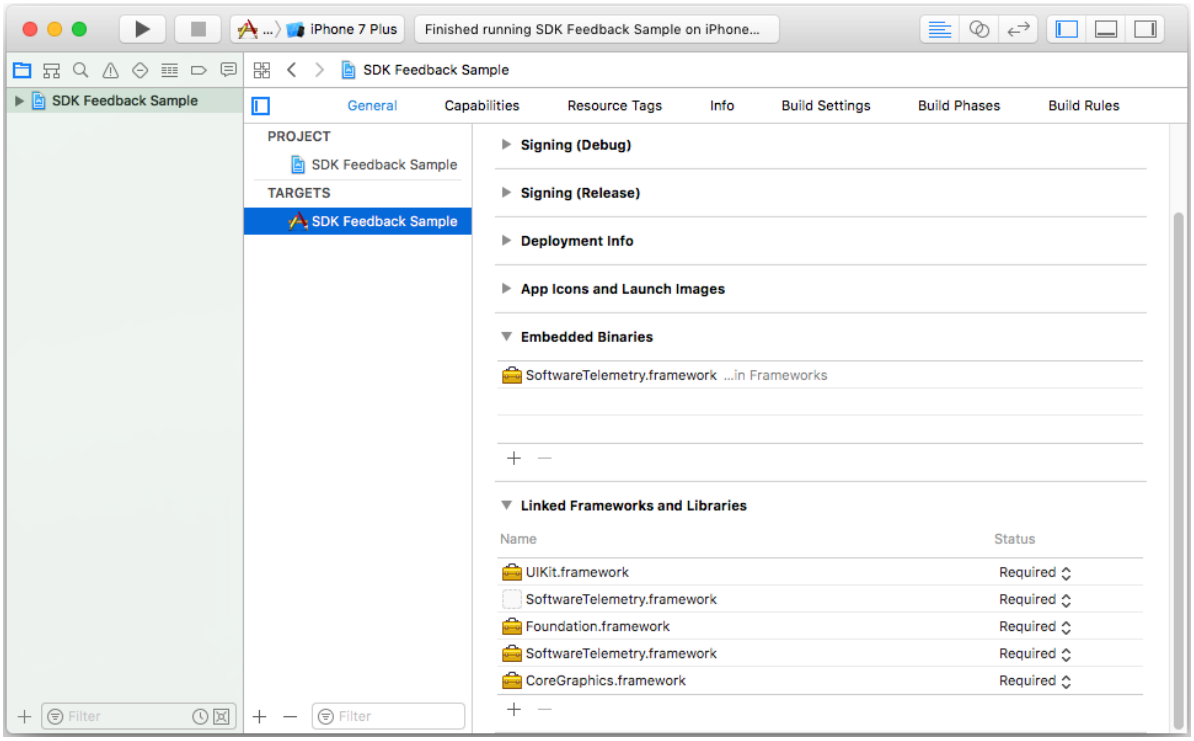
7.1 Prerequisites for Instrumenting iOS Apps

In order to use the Fabasoft app.telemetry SDK for mobile devices (iOS) you will need an installation of Fabasoft app.telemetry (Server, Agent + WebAPI) and ensure that the WebAPI web service is reachable from your mobile device by checking the WebAPI status page of your installation (`https://<your server>/web.telemetry`). This installation will receive the telemetry data and will be used to configure the infrastructure (agent, log pools, etc.).

7.2 Configuration for iOS Apps / XCode

The next required step is to include the Fabasoft app.telemetry SDK library for iOS (binary and header-files) as additional library into your project. All required library files are delivered with the app.telemetry installation media in the "`Developer/iOS`" folder.

Your project has to be configured the following way:



1. Add the SoftwareTelemetry.framework-Framework to the Embedded Binaries and Linked Frameworks and Libraries of your targets general settings
2. Import Software-Telemetry SDK main header file in your project source files:
   - `#import <SoftwareTelemetry/SoftwareTelemetry.h>`

7.3 Instrumenting iOS Apps

The technical background for instrumenting any application is very similar for all languages so for example see chapter 4 “*Java API Reference*”.

As already described for the Android Java SDK (see chapter 6 “*Android API Reference*”) the data transport is HTTP-based and requires a configuration URL to your WebAPI installation on initializing your API for your application. Also most of the methods are thread-based and require a thread-token to be passed as additional parameter.

In order to init the iOS SDK for your mobile app you need to pass the target URL of your app.telemetry WebAPI instance.

Syntax (Initialization)

```
NSString *serverUrl = @" https://<your server>/web.telemetry ";
APM *apm = [APM initializeWithURL:serverUrl];
[apm registerApplicationWithName:@"Fabasoft app.telemetry"
applicationId:@"iOS Sample" applicationTierName:@"Apple iOS SDK"
applicationTierId:@"1"];
```

In the following code snippet is an extract from the sample project listing some of the main SDK functions of the iOS SDK.

Usage Example (`MyViewController.m`)

```
#import "MyViewController.h"
#import <SoftwareTelemetry/SoftwareTelemetry.h>
#import <QuartzCore/QuartzCore.h>   // required for screenshot
@implementation MyViewController
// ...
- (void)viewDidLoad
{
    [super viewDidLoad];
    //change the following URL to YOUR app.telemetry server/WebAPI
    //it is the URL where the telemetry data and the feedback is sent to
    NSString *serverUrl = @" https://<apptelemetry
server>/web.telemetry";
    APM *apm = [APM initializeWithURL:serverUrl];
    [apm registerApplicationWithName:@"Fabasoft app.telemetry"
applicationId:@"iOS Sample" applicationTierName:@"Apple iOS SDK"
applicationTierId:@"1"];
    APMModule *apmMod = [apm getModule:@"TestModule"];
    [apmMod registerEventWithId:900 name:@"Event" parameters:@"text"];
    [apmMod registerEventWithId:901 name:@"send Message"
parameters:@"message"];
    [apmMod registerEventWithId:910 name:@"Test Event" parameters:@""];
    [apmMod registerEventWithId:912 name:@"Event with Params"
parameters:@"p1;p2;p3"];
    // ...
    [self fireSampleRequests:0];
}
- (IBAction)fireSampleRequests:(id)sender {
    APM *apm = [APM instance];
    APMModule *apmMod = [apm getModule:@"TestModule"];
    // ...
    // request 1
    uint64_t ctx = [apm createContextTx:0 filterValue:self.userName];
```

```
    [apmMod eventTx:ctx eventId:900 level:APMLogLevelLog
flags:APMFlagNone stringValue:@"My 1st Event"];
    [apmMod eventTx:ctx eventId:901 level:APMLogLevelLog
flags:APMFlagNone stringValue:greeting];
    [apmMod eventTx:ctx eventId:920 level:APMLogLevelDetail
flags:APMFlagEnter|APMFlagWait stringValue:@"50"];
    usleep(50000);
    [apmMod eventTx:ctx eventId:920 level:APMLogLevelDetail
flags:APMFlagLeave|APMFlagWait];
    [apm releaseContextTx:ctx];
}
- (IBAction)doReport:(id)sender {
    APM *apm = [APM instance];
    APMModule *apmMod = [apm getModule:@"TestModule"];
    uint64_t ctx = [apm createContextTx:0 filterValue:self.userName];
    NSString *repk = [[NSString alloc] initWithFormat:@"REPK_%d",
arc4random_uniform(10000)];
    NSString *nameString = self.textField.text;
    NSString *descr = [[NSString alloc] initWithFormat:@"%@",
nameString];
    [apm reportTx:ctx filterValue:self.userName reportKey:repk
description:descr];
    [apm reportValueTx:ctx reportKey:repk valueKey:@"Rep_Val1"
value:@"Report Value 1"];

    if ([self.switchScreenshot isOn]) {
        [apmMod eventTx:ctx eventId:935 level:APMLogLevelDetail
flags:APMFlagEnter|APMFlagWait];
        APMReportContent *screenshot = [APM
createScreenshotForView:self.view];
        [apmMod eventTx:ctx eventId:935 level:APMLogLevelDetail
flags:APMFlagLeave|APMFlagWait];
        NSString *screenshotSize = [[NSString alloc]
initWithFormat:@"%lu", (unsigned long)[[screenshot content] length]];
        [apmMod eventTx:ctx eventId:931 level:APMLogLevelDetail
flags:APMFlagNone stringValue:screenshotSize];
        [apm reportContentWithKey:repk content:screenshot];
        self.labelResult.text = [[NSString alloc] initWithFormat:@"Report
sent with screenshot: '%@'", descr];
    }
    else {
        [apmMod eventTx:ctx eventId:900 level:APMLogLevelDetail
flags:APMFlagNone stringValue:@"Skipping Screenshot"];
        self.labelResult.text = [[NSString alloc] initWithFormat:@"Report
sent - NO screenshot: '%@'", descr];
    }
    [apm releaseContextTx:ctx];
}
```