OpenPegasus Tracing Development Guide

OpenPegasus provides a tracing facility that helps in investigating the cause of a problem. For example, if requests abort, performance is reduced, or unexpected responses appear, trace can provide pointers to where and when the problem occurred.

To generate a useful trace for problem determination, the trace messages are categorized in trace levels and components. The trace level defines the severity of the message and the trace component assigns the message to a specific module or several modules serving special working units of OpenPegasus and there are special purpose trace components.

Writing trace messages is a OpenPegasus server functionally only. Currently it is not possible for providers to use the tracing facility.

1 When to write trace messages?

Trace statements are for debugging purpose. The information of a trace message is for a developer or service person, a person having insight into the code of OpenPegasus. Trace messages are utilizing the code to enable these people to understand the state of OpenPegasus and print execution flow relevant information like:

- Locations of executions
- Return values of functions
- Error conditions of functions
- Content of variables
- Conditions and it's evaluation
- Etc.

Trace messages are NOT enabled for multicultural support.

Log messages are in contrast to trace messages. Log messages must give system operator/programmer information about the run-time state of OpenPegasus to enable these people to change the runtime environment to work properly.

2 Categorize trace messages

2.1 What is the right trace level?

There are 5 trace levels for configuration but only trace level 1 to 4 can be used for specifying the severity of a trace message.

The trace level definitions are done in **`Pegasus/Common/Tracer.h`**

Trace level 0 is reserved to switch tracing off. Trace level 5 is reserved for method enter/exit statements. Both trace levels cannot be specified explicitly on a trace statement. All other levels can be specified in the tracing macros using the static variables with the definition as listed below.

Level #	Definition
1	Tracer::LEVEL1
2	Tracer::LEVEL2
3	Tracer::LEVEL3
4	Tracer::LEVEL4

Table 1

2.1.1 Tracer::LEVEL1

Trace level 1 should be used for severe error conditions that need to be reported to an user. For instance if the server has to discard data to be able to proceed, throws an exception, or has to close the connection.
All log messages are also written to trace level 1, which means that there is no need to duplicate log messages as a trace message.
As example, at the OpenSSL initialization, not enough seed data is found:

```
PEG_TRACE((TRC_SSL, Tracer::LEVEL1,
    "Not enough seed data in seed file: %s",
    (const char*)randomFile.getCString()));
```

2.1.2 Tracer::LEVEL2

Trace level 2 is for basic flow trace messages with minimal detail. These messages should also report error conditions which might cause an error visible to the end user, but don't necessarily have to.
For example, after reading an HTTP request, OpenPegasus is not able to parse the content language:

```
PEG_TRACE((TRC_HTTP, Tracer::LEVEL2,
    "HTTPConnection: ERROR: contentLanguages had "
         "parsing failure. clearing languages. error data=%s",
    (const char*)contentLanguagesString.getCString()));
```

2.1.3 Tracer::LEVEL3

Trace level 3 is for intra-function logic flow and moderate data detail. These messages should report details of decisions made and important data of the general flow.
For example, after a connection has been established, the client IP address is written:

```
PEG_TRACE((TRC_HTTP, Tracer::LEVEL3,
    "Connection IP address = %s",(const char*)_ipAddress.getCString()));
```

2.1.4 Tracer::LEVEL4

Trace level 4 is the highest data detail. These messages should report everything needed to be able to follow the flow.
For example:

```
PEG_TRACE((TRC_PROVIDERMANAGER,Tracer::LEVEL4,
    "CMPIProvider has pending operations: %s",
    (const char*)provider->getName().getCString()));
```

2.2 What is the right trace component?

At each trace macro, a trace component ID has to be specified to pool the trace messages in domains. These domains can be one or several modules serving a dedicate purpose within OpenPegasus or a special purpose domain serving an overall purpose.

The trace components ID's used for the trace macros are defined as enumeration in **`Pegasus/Common/Tracer.h`**.

The TRACE_COMPONENT_LIST[] used for the configuration option is defined in **`Pegasus/Common/Tracer.cpp.`**

The components to be specified at the traceComponents configuration are defined in the TRACE_COMPONENT_LIST[]. If a new trace component is needed or changed, you have to modify both lists at the same index.

The table below is a first reference for component ID's for dedicated functionality of OpenPegasus. The current definitions can be found in **`Pegasus/Common/Tracer.h`**

enum TraceComponentID	**static char const* TRACE_COMPONENT_LIST[]**
TRC_AUTHENTICATION	Authentication
TRC_AUTHORIZATION	Authorization
TRC_EXP_REQUEST_DISP	CIMExportRequestDispatcher
TRC_CIMOM_HANDLE	CIMOMHandle
TRC_CMPIPROVIDER	CMPIProvider
TRC_CMPIPROVIDERINTERFACE	CMPIProviderInterface
TRC_CQL	CQL
TRC_CONFIG	Config
TRC_CONTROLPROVIDER	ControlProvider
TRC_DISPATCHER	Dispatcher
TRC_EXPORT_CLIENT	ExportClient
TRC_HTTP	Http
TRC_INDICATION_GENERATION	IndicationGeneration
TRC_IND_HANDLER	IndicationHandler
TRC_INDICATION_RECEIPT	IndicationReceipt
TRC_INDICATION_SERVICE	IndicationService
TRC_L10N	L10N
TRC_LISTENER	Listener
TRC_MESSAGEQUEUESERVICE	MessageQueueService

TRC_OBJECTRESOLUTION	ObjectResolution
TRC_OS_ABSTRACTION	OsAbstraction
TRC_PROVIDERAGENT	ProviderAgent
TRC_PROVIDERMANAGER	ProviderManager
TRC_REPOSITORY	Repository
TRC_SSL	SSL
TRC_SERVER	Server
TRC_SHUTDOWN	Shutdown
TRC_THREAD	Thread
TRC_USER_MANAGER	UserManager
TRC_WQL	WQL
TRC_WSMSERVER	WsmServer
TRC_XML	Xml
TRC_XML_IO	XmlIO

Table 2

2.3 Special purpose trace components

The next table below specifies component ID's for special purpose of the OpenPegasus:

Special purpose TraceComponentID	Description
TRC_DISCARDED_DATA	Issues a trace message when information is discarded or an operation is cancelled to enable OpenPegasus to proceed. Tracer::LEVEL1 must be used for this component.
TRC_LOGMSG	Do not use this trace component. All messages written by the Logging Facility are automatically written with this component.
TRC_XML_IO	This component specifies trace statements which belongs to the XML request/response handling and should be used only in this context.

3 The Trace Interface

There are 4 macros available to write a trace:

Trace macros	Description
PEG_METHOD_ENTER	logs a method entry message to the trace.
PEG_METHOD_EXIT	logs a method exit message to the trace file.
PEG_TRACE_CSTRING	This writes a single trace message.
PEG_TRACE	Writes a printf()-style formatted trace message

All macros automatically add file (__FILE__) and line number (__LINE__) to the trace message.

You have to include <Pegasus/Common/Tracer.h> to use tracing at all.

Special note on using type String

If you like to use a String object as parameter in a trace macro, you have to

1. get the CString using the method getCString()
2. cast to (const char*)

For example:

```
PEG_TRACE((TRC_HTTP, Tracer::LEVEL2,
    "HTTPConnection: ERROR: contentLanguages had "
         "parsing failure. clearing languages. error data=%s",
    (const char*)contentLanguagesString.getCString()));
```

3.1 PEG_METHOD_ENTER

SYNOPSIS

```
PEG_METHOD_ENTER (<traceComponent>, char* methodName);
```

DESCRIPTION

<traceComponent>	Refers to a trace component specified in Table 2.
MethodName	The name of the method being entered.

PEG_ METHOD_ENTER() must be placed right after the entry point of a method or function and is generating a method enter trace statement with level 5.

EXAMPLE

```
PEG_METHOD_ENTER(TRC_SSL,"SSLContextManager::createSSLContext()");
```

3.2 PEG_METHOD_EXIT

SYNOPSIS

```
PEG_METHOD_EXIT();
```

DESCRIPTION

PEG_ METHOD_EXIT() must be placed before any exit point of a method or function and is generating a method exit trace statement with level 5.

It can only be used if the PEG_METHOD_ENTER() was used before in the same method/function.

EXAMPLE

```
void SSLContextManager::createSSLContext( … )
{
    PEG_METHOD_ENTER(TRC_SSL,"SSLContextManager::createSSLContext()");

    try
    {
    }
    catch(…)
    {
       PEG_METHOD_EXIT();
       throw MyException();
    }

   PEG_METHOD_EXIT();
   return;
}
```

3.3 PEG_TRACE_CSTRING

SYNOPSIS

```
PEG_TRACE_CSTRING(<traceComponent>,<traceLevel>, const char* traceMessage);
```

DESCRIPTION

<traceComponent>	Refers to a trace component specified in Table 2.
<traceLevel>	Refers to a trace level Tracer::LEVEL1 to Tracer::LEVEL4.
traceMessage	Single trace message.

This writes a single trace message at a certain trace level and component to the trace. The message must be a single character string with a trailing '\0'.

EXAMPLE

```
PEG_TRACE_CSTRING(TRC_HTTP, Tracer::LEVEL1,
    "select() timed out waiting for the socket connection to be"
        "established.")
```

3.4 PEG_TRACE

SYNOPSIS

```
PEG_TRACE((<traceComponent>,<traceLevel>, const char* formatString,……));
```

DESCRIPTION

<traceComponent>	Refers to a trace component specified in Table 2.
<traceLevel>	Refers to a trace level Tracer::LEVEL1 to Tracer::LEVEL4.
formatString, ……	The printf()-style format string and it's parameters.

PEG_TRACE() writes a printf()-style formatted trace message at a certain trace level and component to the trace.

Note:

The parameters have to be put into a single argument for PEG_TRACE() by putting them in additional parenthesis.

EXAMPLE

```
PEG_TRACE((TRC_HTTP, Tracer::LEVEL4,
    "Connection to server in progress.  Waiting up to %u milliseconds "
         "for the socket to become connected.",
    timeoutMilliseconds));
```

4 Compiling out Trace code from Pegasus build

The trace code can be optionally removed from Pegasus at build time by defining PEGASUS_REMOVE_TRACE (–D option).