# PASOE Memory Monitoring

**Enabling tooling to aid debugging PAS instance memory usage.**

March 2019

# Summary

This guide should help to answer the question “*How can I debug memory issues with my PAS environment?*” with regards to the **Progress Application Server for OpenEdge (PASOE)** in versions **11.7.4 and later**. During development, projects are typically tested through unit tests and QA processes. However, these tasks do not always include memory profiling nor consist of long-running processes to detect potential leaks or application performance issues at a server level.

In the following content we will first establish the factors which affect PASOE performance and the importance of monitoring certain metrics. Once these points have been established we will outline the process for automating the collection of these metrics using the “Diagnostic.pl” from the Spark-Toolkit which can then be parsed for trending and further analysis.

Through use of the Diagnostic.pl library you can add some automatic logging of information about your PAS environment. This will utilize an activate and deactivate procedure to perform actions at a per-request level (or possibly the startup and shutdown procedures for session-level monitoring). The included items which can be readily tracked include session information: active state, memory consumption; objects in ABL memory: buffers, handles, temp-tables, class objects, etc.; requests to the session: last 1,000; and profiler output: triggered by execution time threshold. These metrics can be enabled or disabled via a simple JSON config file and can be configured to run during certain times of day over multiple days.

# Prerequisites

The following material **requires use of OpenEdge 11.7.4** as this release provides the expected monitoring tools as part of the OEJMX (Java) and OEManager (API) endpoints. Use of the diagnostic suite with OE versions prior to this service pack is not possible. Note: Much of the following material should be applicable to the OpenEdge 12 release as well. You will need the latest [Diagnostic.pl](https://github.com/progress/Spark-Toolkit/blob/master/dist/Diagnostic.pl) library as found in the open-source [Spark-Toolkit](https://github.com/progress/Spark-Toolkit) project repository on GitHub **using release 4.4.2 or later**.

# Memory Management in PASOE

It is worth exploring a bit more about how a PASOE instance utilizes memory and how a steady “leak” from application code can contribute in a profound way, leading to unexpected behavior.

First, we must establish that PAS uses a Multi-Session AppServer (or MSAS) Agent to manage the ABL runtime “threads” which services each request. Due to this multi-session nature of the MSAS Agents, issues such as unregulated memory usage can have a multiplying effect on the overall agent process.

**Example:** If a PAS instance is configured with 1 agent and 20 sessions per agent, and each session requires an average of 30MB (for managing ABL objects, HTTP requests/responses, etc.) then the overall agent would need 20 \* 30MB of memory, or 600MB. Now let’s say the agent had 40 or 60 sessions which would mean 1.2GB or 1.8GB. And if the amount of memory per session expands beyond our 30MB example, this would likewise increase the total needed by the agent over time.

Next, it should be noted that memory as consumed by the MSAS Agent is subject to management by the operating system. Typically, it should be expected that as processes consume memory they *may* not immediately return unused (freed) memory back to the OS. In the case of the current PAS architecture, much of the memory allocated to an MSAS Agent becomes associated with that process until the process is terminated. As such, the sessions within each agent may take and give memory as necessary, but that memory is still allocated to the parent MSAS Agent process. Therefore, it is crucial to assist the AVM’s garbage collection by deleting any unused ABL objects, buffers, procedures, temp-tables, handles, etc. during code execution to ensure sessions do not grow their memory footprint any more than necessary.

Finally, in the event that an MSAS Agent consumes too much memory, one of many OS-defined actions may occur. As an example, for many Linux distributions when available system memory reaches a determined low-point, a watchdog process called the “Out-of-Memory Killer“ (or OOM) may be called upon to identify and terminate any processes which are consuming the most memory. Naturally this may be an \_mproapsv (MSAS Agent) process which is servicing multiple requests via its sessions. In our previous example case this could be 20, 40, 60, or any number of sessions which will be abruptly terminated if the process is killed.

**NOTE:** You can determine if a process was terminated by the OOM killer by checking the system logs (eg. /var/log/messages or /var/log/dmesg) and grep for “killed process”. This should provide a timestamped entry and PID information about when the process was terminated.

In Comparison to Classic AppServer

The Classic AppSserver’s Agent processes (Classic WebSpeed included) contained a single ABL Session, the multi-session Agent contains many. Any ABL Session, whether in a single-threaded Agent process or a multi-session Agent process, will consume the same amount of OS process memory given the execution of the same r-code. They both will release ABL Session allocated OS process memory on the same schedule. This makes sense since they both employ the same core AVM language engine. The difference at the OS level is not that one application server uses more memory than another, but in the visibility of where the memory is used: by one OS process or distributed across many.

Likewise, an ABL application r-code memory leak will be the same in both a classic and multi-session Agent process. Cumulatively at the OS level the same amount of OS memory will be consumed and leaked—the difference being whether it is visible in one OS process or distributed across multiple OS processes.

# Monitoring Metrics

As we begin to explore the various metrics available for monitoring, we need to establish what we are looking for and how that affects the choice of metrics. Typically there are 2 situations where we may wish to observe behavior of the server: performance and stability.

In terms of performance, this could be requests which appear to be taking an unusually long time to execute. Factors which may contribute to this behavior may include a problematic function, method, or procedure which may be called more times than expected within the code. To diagnose this the optimal metric is the code profiler. Luckily this feature can be enabled at runtime (at the start of a request or session) and can be generated on-demand if necessary (at the end of a request or session). Another metric which could be useful is to capture the ABL execution time for the request and compare that to the response time as reported by Tomcat. Any discrepancy between elapsed times as reported by the Tomcat access log and the ABL code execution may indicate a queuing/waiting situation in waiting for an available session, or an issue in returning data from the session through the Tomcat web server. eg. Tomcat reports a request took 1 second though the ABL execution only took only 100ms, which may indicate some kind of unexpected waiting at the web server level.

In terms of stability, the most common culprit is memory usage and can be easily diagnosed by use of 2 internal metrics. The first is the **ABLObjects** report, which must run at the very end of a request to determine if the code was appropriately cleaned up. Any items which appear in this report may be subject to further scrutiny. The other half of this combo is the **Session** information for each MSAS Agent. This data reflects the current memory consumption for each session, and when graphed over time can indicate a trend in memory use. ***Typically, an increase in memory use is seen with an increase in the number of ABLObjects.***

# Available Tools

As of the release of 11.7.2, new features in OpenEdge have provide some out-of-the-box means of monitoring behavior of your PASOE instance. These have been improved and solidified through the subsequent SP3 and SP4 releases. At present there are 2 primary means of accessing the same information about your running instances:

**OEJMX Queries** – Operates at the OS level, allowing you to query the running Tomcat Java process. This can be run via shell/batch scripts in an unobtrusive manner, even in a production environment.

A natural drawback to this approach is that the ABLObjects report will reflect ALL objects in memory at the point in time when the OEJMX command is executed—which could capture valid objects “in flight” during execution of ABL code and may not be accurate for purposes of identifying a leak. The only exception to this would be if it is known that there is no user activity against any of the MSAS Agents, when all sessions are shown as in an IDLE state.

**OEManager API’s** – Utilizes RESTful requests to the oemanager WebApp as found in non-production PASOE instances (installed when using the -f option for “pasman create”). This can also be installed after server creation on production-class servers, such as QA/UAT or Staging type environments.

This approach can be utilized via the REST-Out (or OEHttpClient) feature in the ABL and allows for the collection of metrics to be run at a specific time. Since this is controllable within the code of the application, it can be executed at request boundaries. This is why use of the “deactivate” procedure is recommended as it will be run when a session/request is completed.

**Note:** Due to the nature of certain queries’ ability to stop an instance, use of the OEManager is not recommended for production environments (or at least not without changes to security such as altering the default password and whitelisting of allowed IP addresses).

# Mitigating Server Downtime

There are a few practices and features which can allow you to better manage your PASOE instances before they are suddenly handled by the OS. This is by no means a substitute for good coding practices, though we’ll explore that aspect later in this document. The purpose here is to define a means of manually controlling your PAS instance to avoid abrupt shutdowns and maximize runtime/availability.

Hot Agent(s) + Spare Agent Configuration

Not a feature, per se, but a pattern for configuring your PASOE instance as based on expected capacity. Ideally you should be able to run the necessary number of sessions on a single MSAS Agent, though in reality that can differ by application, scaling architecture, and expected traffic. In this scenario, ideally you would have N agents to handle all requests with one extra agent running but not utilized (N+1).

In the event an agent is consuming too much memory or needs to be terminated, the idea is that you can issue a graceful shutdown of an agent your configuration will cause traffic to shift to the unused agent while another starts up. For a simple example of 1 agent with 1 spare, your openedge.properties file would consist of the following options:

minAgents=2

maxAgents=2

numInitialAgents=2

As noted, in order to keep at least 1 agent as a “spare” then the **maxConnectionsPerAgent** should be set sufficiently for the expected number of concurrent requests to the PAS instance. This should allow the first agent to handle the load on the server while leaving the second agent relatively idle.

Graceful Agent Shutdown

If an agent is nearing a tipping point or needs to be shut down, 11.7.3 introduced options to stop an agent in a more graceful manner: **waitToFinish** and **waitAfterStop**. This allows existing connections to complete while preventing further requests from being sent to the agent. For more information about how to utilize these options, please see the resolution details in [KB Article 000089066](https://knowledgebase.progress.com/articles/Article/trim-pasoe-closes-connection-7243-7241-000089066).

# Using the Spark Diagnostic Library

Within the Spark-Toolkit repository’s /dist/ folder is a suite of procedures and classes within the **Diagnostic.pl** procedure library. These are separate from the Spark.pl library and do not require the latter to be included in your project. The purpose of these tools is to automate the available OEManager API’s and/or OEJMX queries and produce output in a consistent way for easy parsing. In particular, some monitoring (eg. ABLObjects) must be done at specific times within your ABL code in order to get accurate results. As such, this code is meant to run within an application on a non-production PASOE instance.

Deploying the Tooling (Overview)

1. Confirm your OpenEdge version to be **11.7.4** or later.
   1. Do not proceed if you are running an earlier version!
   2. If using OE 11.7.x, please use the PL from /dist/oe11/
   3. If using OE 12.0, please use the PL from /dist/oe12/
2. Copy the **Diagnostic.pl** to CATALINA\_BASE/openedge
3. Add **Diagnostic.pl** to your ABL Application’s PROPATH in openedge.properties
4. Add a **metrics\_config.json** to /bin directory.
   1. Edit options as necessary for your environment (see next section).
5. Add a **metrics\_setenv.[sh|bat]** to /bin directory.
   1. Edit variables as necessary for your environment.
6. Add **sessionActivateProc** and **sessionDeactivateProc** to openedge.properties
   1. Or, add “RUN” statement to any existing Activate/Deactivate procedures.
7. Add **sessionStartupProc** and **sessionShutdownProc** to openedge.properties
   1. Or, add “RUN” statement to any existing Startup/Shutdown procedures.
8. Copy **logging.config** to CATALINA\_BASE/openedge

Sample metrics\_config.json

This file drives the operation of the metrics-gathering within the diagnostic logic. The file consists of 3 sections which relate to specific behavior: “general” options, “remote” requests for metrics, and ABL “profiler” output. For reference, below is a sample of the JSON content and some initial values for the options with each section and its options described in more detail.

{

"general": {

"pollIgnore": "",

"pollStart": "2018-12-01T08:00:00",

"pollStop": "2018-12-31T20:00:00"

},

"remote": {

"exclude": "\*Spark.\*",

"oemUser": "tomcat",

"oemPass": "tomcat",

"outputInterval": 120000,

"source": "oemanager",

"trackMemory": **false**,

"trackObjects": **false**,

"trackRequests": **false**

},

"profiler": {

"enabled": **false**,

"filter": "\*",

"threshold": 1000,

"trackBy": "request"

}

}

**General**

**pollIgnore (String)** – Allows you to configure a comma-delimited list of classes to ignore when polling. Examples of this may include any classes related to OERealm authentication as this should not affect system performance and we do not want to delay this process when it is necessary to gather metrics.

**pollStart/pollEnd (ISO8601 Date String)** – Defines a window in which polling can take place to start and stop polling, respectively.

**Remote**

**source (String)** – Default should be “oemanager” which defines the remote source of metrics and indicates that requests will be sent via HTTP to the RESTful OEM-API endpoints.

**oemUser (String)** – Username for the OEM webapp. Default is “tomcat”.

**oemPass (String)** – Password for the OEM webapp. Default is “tomcat”.

**outputInterval (Integer)** – Time in milliseconds between any requests to the remote endpoints to query for any tracked metrics. Note: Set to a large interval (1 hour or greater) if intending to run diagnostics for more than a few days.

**trackMemory (Boolean)** – Indicates whether the MSAS Agent Session information should be tracked, which includes a snapshot of memory consumed by each session at that point in time.

**trackObjects (Boolean)** – Indicates whether a list of ABLObjects in memory should be tracked and reported. This data should correspond to memory usage and potentially indicate any leaks.

**exclude (String)** – A comma-delimited list of class packages to ignore while tracking ABLObjects. Wildcard matches are allowed, and will allow you to skip entire groups of objects. In the case above, skipping \*Spark.\* objects will focus solely on the application code rather than the infrastructure/toolkit classes (eg. the diagnostic classes themselves).

**trackRequests (Boolean)** – Indicates whether each request to an ABL runtime should be tracked. This is useful for debugging the time each request takes to execute within the MSAS Agent Session, which can be compared to the request time reported by Tomcat in the localhost\_access log file. Unexpected overhead may indicate a queing/waiting scenario between the web server and the ABL Application.

**Profiler**

**enabled (Boolean)** – Turns on the ABL Profiler to track code execution at a session or request level. As performance can be affected by calling code too often, use of the ABL Profiler can help identify this scenario or perhaps long-running procedures.

**trackBy (String)** – Valid values are “session” or “request” to determine if the profiler should be enabled for either an entire MSAS Agent **session** or each individual **request**. If tracking by **session**, the profiler will be started when each session starts and will always stop and dump results when each session shuts down. If tracking by request, the profiler will be started and stopped with each request, while the filter and threshold values will be used to determine if output should be created.

**filter (String)** – A comma-delimited list of programs (classes, procedures, etc.) to specifically watch during reporting by “request”. By default, use of “\*” will report all items. Ignored when tracking by “session”.

**threshold (Integer)** - If tracking by **request**, if the elapsed execution time exceeds the given value in milliseconds then the code will be profiled and dumped to a specific directory to log that request (otherwise, no output will be created). Ignored when tracking by “session”.

Sample metrics\_setenv.bat

For Windows environments, use the following to specify the ABLApplication name to be tracked, and the relative URI where the oemanager WebApp may be found for this instance.

@echo off

rem Set the name of the ABLApp which you want to monitor.

set ABLAPP\_NAME=oepas1

rem Set the URL of the server where we can reach the OEManager webapp.

set INSTANCE\_URI=http://localhost:8810

rem Set the name of the config file to use at runtime.

set METRICS\_CONFIG=metrics\_config.json

exit /b 0

Sample metrics\_setenv.sh

For Linux/Unix environments, use the following to specify the ABLApplication name to be tracked, and the relative URI where the oemanager WebApp may be found for this instance.

#!/bin/sh

# Set the name of the ABLApp which you want to monitor.

ABLAPP\_NAME=oepas1

export ABLAPP\_NAME

# Set the URL of the server where we can reach the OEManager webapp.

INSTANCE\_URI=http://localhost:8810

export INSTANCE\_URI

# Set the name of the config file to use at runtime.

METRICS\_CONFIG=metrics\_config.json

export METRICS\_CONFIG

Sample logging.config

As part of the solution to tracking data on an Agent and Session level, output must be carefully logged in a clear and consistent manner. Output generated by the diagnostic code will be appropriately named and dumped to the **session temp directory** under a “**metrics**” folder. And through use of the service-level logging feature in OpenEdge we can specify a pattern for naming and output of any messages derived from the monitoring logic itself. This is useful for debugging the debugging to determine if things are running as expected. The contents below will generate information for each session of each agent, placing the content within the same “metrics” folder used by the remainder of the diagnostic code.

{

"logger": {

"Spark.Diagnostic.Util.RemoteMetrics": {

"logLevel": "INFO",

"filters": [

"ABL\_SUBSTITUTE\_FORMAT",

"ERROR\_FORMAT",

{

"name": "TOKEN\_FORMAT",

"format": "[${t.now}] ${req.id} | ${msg.logger}:${msg.level} - ${msg}"

},

{

"name": "NAMED\_FILE\_WRITER",

"fileName":

"${session.temp-dir}/metrics/Actions-A${req.agent}-S${req.session}.log",

"appendTo": true

}

]

},

"Spark.Diagnostic.Util.OEMetrics": {

"logLevel": "INFO",

"filters": [

"ABL\_SUBSTITUTE\_FORMAT",

"ERROR\_FORMAT",

{

"name": "TOKEN\_FORMAT",

"format": "[${t.now}] ${req.id} | ${msg.logger}:${msg.level} - ${msg}"

},

{

"name": "NAMED\_FILE\_WRITER",

"fileName":

"${session.temp-dir}/metrics/OEMetrics-A${req.agent}-S${req.session}.log",

"appendTo": true

}

]

}

}

}

Modifying openedge.properties and Procedures

In order to execute the necessary diagnostic code, changes to both the PASOE configuration and related procedure files are currently necessary. To ensure all features can be enabled as designed the following instructions will address all options which need to be added/modified for operation.

**Updating PROPATH** – Adds the diagnostic libraries to the application.

1. Copy the **Diagnostic.pl** to your CATALINA\_BASE/openedge directory.
2. Locate the **[AppServer.Agent.<abl\_app\_name>]** block for your ABL Application.
3. Add the entry **${CATALINA\_BASE}/openedge/Diagnostic.pl** to the PROPATH.

**Session Startup** – Starts the profiler across a session.

1. Locate the **[AppServer.Agent.<abl\_app\_name>]** block for your ABL Application.
2. Check if a **sessionStartupProc** exists within this block:
   1. If yes, skip to the instructions for modifying an existing procedure.
   2. If no, add **sessionStartupProc=Spark/Diagnostic/metrics\_startup.r**
3. If an existing procedure is defined, note the path to the .p as we should not modify any existing application code with our changes. Create a new procedure somewhere in the PROPATH of the application called “**custom\_startup.p**”. Add the following code to the new procedure, replacing the “<original\_procedure>” name below as necessary:

define input parameter startup-data as character no-undo.

run Spark/Diagnostic/metrics\_startup.

run <original\_procedure> ( input startup-data ).

Finally, add **sessionStartupProc=custom\_startup.p** to the config file.

**Session Activate** – Starts per-request monitoring.

1. Locate the **[AppServer.Agent.<abl\_app\_name>]** block for your ABL Application.
2. Check if a **sessionActivateProc** exists within this block:
   1. If yes, skip to the instructions for modifying an existing procedure.
   2. If no, add **sessionActivateProc=Spark/Diagnostic/metrics\_activate.r**
3. If an existing procedure is defined, note the path to the .p as we should not modify any existing application code with our changes. Create a new procedure somewhere in the PROPATH of the application called “**custom\_activate.p**”. Add the following code to the new procedure, replacing the “<original\_procedure>” name below as necessary:

run Spark/Diagnostic/metrics\_activate.

run <original\_procedure>.

Finally, add **sessionActivateProc=custom\_activate.p** to the config file.

**Session Deactivate** – Ends per-request monitoring.

1. Locate the **[AppServer.Agent.<abl\_app\_name>]** block for your ABL Application.
2. Check if a **sessionDeactivateProc** exists within this block:
   1. If yes, skip to the instructions for modifying an existing procedure.
   2. If no, add **sessionDeactivateProc=Spark/Diagnostic/metrics\_deactivate.r**
3. If an existing procedure is defined, note the path to the .p as we should not modify any existing application code with our changes. Create a new procedure somewhere in the PROPATH of the application called “**custom\_deactivate.p**”. Add the following code to the new procedure, replacing the “<original\_procedure>” name below as necessary:

run Spark/Diagnostic/metrics\_deactivate.

run <original\_procedure>.

Finally, add **sessionDeactivateProc=custom\_deactivate.p** to the config file.

**Session Shutdown** – Ends profiler monitoring of a session.

1. Locate the **[AppServer.Agent.<abl\_app\_name>]** block for your ABL Application.
2. Check if a **sessionShutdownProc** exists within this block:
   1. If yes, skip to the instructions for modifying an existing procedure.
   2. If no, add **sessionShutdownProc=Spark/Diagnostic/metrics\_shutdown.r**
3. If an existing procedure is defined, note the path to the .p as we should not modify any existing application code with our changes. Create a new procedure somewhere in the PROPATH of the application called “**custom\_shutdown.p**”. Add the following code to the new procedure, replacing the “<original\_procedure>” name below as necessary:

run Spark/Diagnostic/metrics\_shutdown.

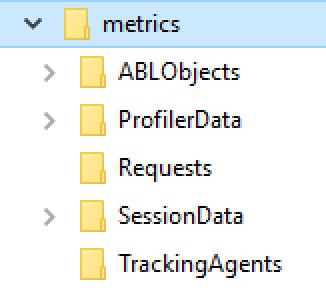
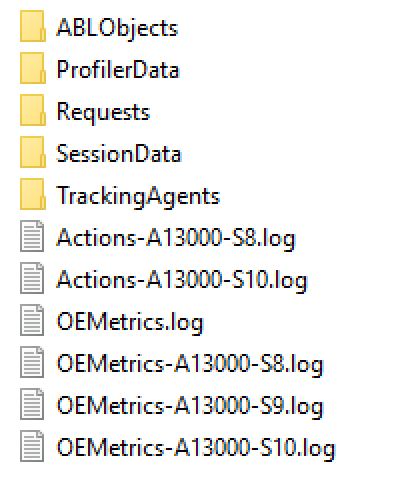
run <original\_procedure>.

Finally, add **sessionShutdownProc=custom\_shutdown.p** to the config file.

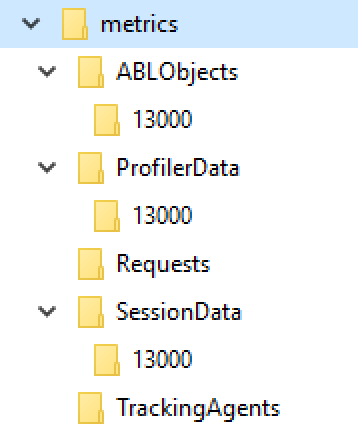
Once the openedge.properties file is updated, and all scripts/configs are placed in their respective locations, you can stop and start the PASOE instance to pick up the changes. If metrics are set to be gathered immediately (based on the provided polling start/stop timestamps) results should begin to appear within the default location **<session\_temp\_dir>/metrics** nearly immediately. Depending on the polling interval it may take some time to begin to see the ABLObjects and Sessions data being reported.

Examining Output

When examining the resulting metrics folder, the following standard directories should be present: ABLObject, ProfilerData, Requests, SessionData, and Tracking Agents. The root folder for metrics also contains debug logs for the operation of the diagnostics. The “OEMetrics” files reflect information about the diagnostics overall, while the “Actions” logs contain specific messages about requests to the OEManager webapp endpoint. The verbosity of these logs can be managed via the service-level logging configuration file (logging.config). Each log file should be separated by the MSAS Agent PID and Session ID to which they belong. This is done to ensure that a high-transaction server does not cause file locking/contention by attempting to write to the same common log file.



Expanding the directories should show additional directories for each MSAS Agent PID as available to the PASOE instance at the time of logging. In the case below, a PID 13000 is running for the instance and we are tracking the ABLObjects, sessions (memory) and profilter output. Each PID subdirectory should then contain only JSON files with the data as collected by the diagnostic tooling. The filename should consist of a timestamp in ISO8601 format and identifier for the Agent PID and optionally a Session ID (SessionData reports all sessions for an agent, and so this data is not separated into separate files). This structure should make it easy to parse the results either by human or machine and know specifically when and by what process the data was produced.



TODO: Document the “PAS Monitor” solution for automating the intake and processing of results. This requires the publication of another application which is not yet ready for public consumption.