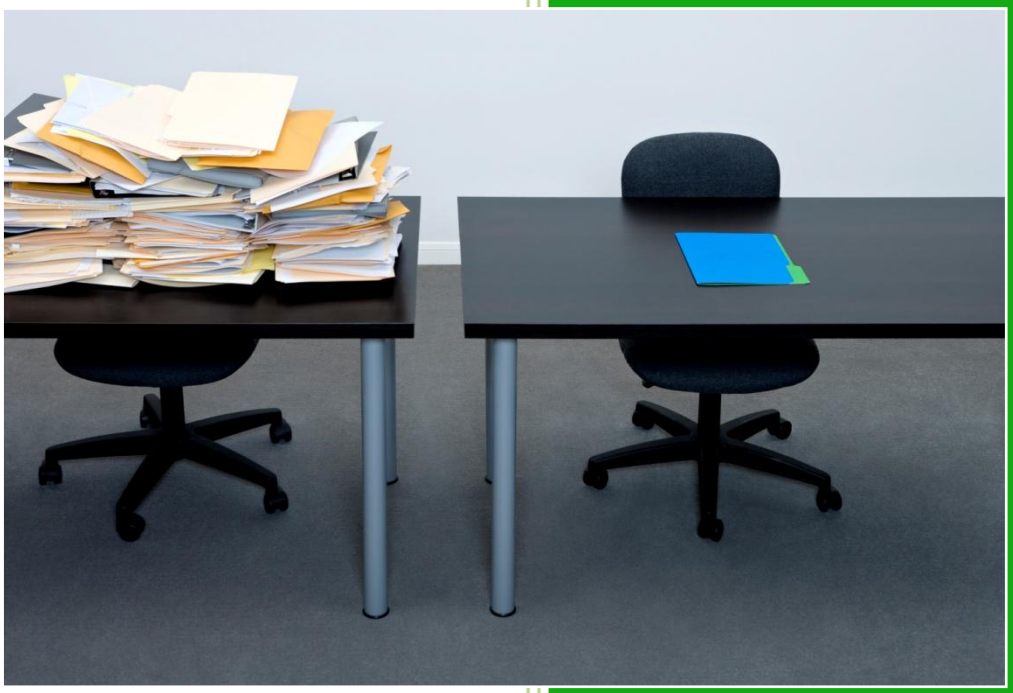
Clarity Technical Implementation Guidelines



Contents

Version: **1.0.4**

Revision Date: **13 June 2011**

---

1 Intended Audience

This document is intended for CA Technology employees, partners, and customers who are implementing technical configurations, integrations, and customizations in a Clarity PPM. As of the writing of this guide, Clarity 12.1.0 is the most recent release. Adjustments for subsequent releases may be necessary.

2 Executive Summary

If you only have 15 seconds to read this document, this is what you should know:

- Configurations and customizations are governed by the "Customization, Modification, and Configuration Policy" document which can be found at:

  https://support.ca.com/phpdocs/7/5590/5590_Configuration_Policy.pdf

- "Configurations" are the use of documented features to modify the behavior of the as-shipped product, e.g. portlets, processes, reports, NSSQL, custom objects, etc.
  - Clarity has few governing controls on the use of these features, so misuse or aggressive use can affect performance/stability. It is the responsibility of the Services team or the customer to fix these performance issues created by poor configurations.
  - Follow guidelines in this doc where possible to prevent issues.
- "Customizations" are the use of undocumented means to modify product behavior, e.g. Java code, stored procedures, triggers, etc.
  - Not covered by standard support or maintenance contracts.
  - Follow guidelines outlined in this document.
  - Data volume or complexity may require infrastructure enhancements.

3 Configuration versus Customization

Configurations and customizations are governed by the "Customization, Modification, and Configuration Policy" document which can be found at:

https://support.ca.com/phpdocs/7/5590/5590_Configuration_Policy.pdf

3.1 Configurations

Configurations are defined as the use of documented features to modify the behavior of the as-shipped product. Examples of documented features that can be used to create configurations are:

- Process flows
- Clarity Studio portlets
- XOG-based integrations
- Any UI based change

Documented features are supported by CA but the related configurations themselves are not. As an example, the Clarity Studio feature used to build a graph portlet is supported, but the particulars of the underlying graph portlet NSQL code, which is authored to supply underlying data to the portlet, is not. In these situations CA will verify that documented features are operating as expected given the specific configuration change, but CA will not troubleshoot or be responsible for the configuration itself. The developer must be responsible for the design and optimization of their customizations to assure proper performance and reliability.

Examples of configuration changes or custom content that is not supported:

- NSQL
- GEL scripts
- Custom Business Objects content including reports and universes

- Custom XOG integrations

CA will work with the customer to isolate problems experienced when using documented features to confirm that they are a result of incorrectly performing product features (vs incorrectly designed or implemented configurations).

Some configurable features in Clarity can result in performance problems if special consideration is not made when rolling them out. Those features are fully supported and should follow the best practice guidelines in this document.

3.2 Customizations

Customizations are the use of undocumented means to modify product behavior. Example customizations are database schema additions, triggers, and code modifications. These are direct alterations to the server that is used to deliver the Clarity application.

Customizations specifically include:

- Changes to any file that is not included in an emergency bug fix (EBF), patch or upgrade.
- Changes to any database object in the Clarity schema.
- Interfaces developed via Java or other programming languages. This includes any code being executed on the Clarity servers that is not a part of the out of the box product.

There are no mechanisms within Clarity or the environment to limit or govern the behavior of such changes or additions to the Clarity offering.

4 Configuration Guidelines

Clarity's flexibility requires special care be taken when implementing certain features. The following document areas of Clarity that need to be planned carefully before rolling out into production. If not properly configured or tested these features can result in possible performance problems with the system.

We cover best practices for the following configurations in this guide:

- NSQL
- Portlet Design
- GEL Scripting
- Business Objects Reporting
- Process Design
- Implementing Auditing
- Changes to Master Calendars
- Rate Matrix Extraction
- Timeslicing and Datamart

4.1 NSQL

NSQL is the SQL scripting language used for Clarity portlets. Please reference the official Clarity PPM Integration guide on how to utilize the NSQL language.

When writing NSQL, the following best practices should be applied:

- Analyze execution plans for queries that you develop for either portlets or dynamic lookups – ensure that queries perform optimally
- When there is a danger of a large number of rows being returned, consider using "ROWNUM < n" for Oracle or "SELECT TOP n" in SQL Server to prevent users pulling back too much data
- All NSQL portlet queries should contain comment blocks, detailing the use, author and source portlet.

4.2 Portlet Design

Portlets are used to create custom Clarity pages. Portlets are not a replacement for proper reports. The Clarity PPM Studio Guide describes the Architectural Design and Intent of a portlet. In summary:

- Portlets are snapshots into CA Clarity PPM data and can consist of grids, graphs, or snippets of HTML.
- While portlets do not replace CA Clarity PPM reports, they can be regarded as mini-reports.
- You can create and publish portals across the enterprise. Each portal page is comprised of a set of portlets – small windows of information presented as graphs, tables or web page snippets.

When creating portlets, the following best practices should be applied:

- When utilizing aggregation over many rows of data, it is a best practice to create a separate summary portlet (e.g., one line portlet with totals.) A summary portlet allows the aggregation to be done on the database instead of returning all rows to the grid portlet to calculate the aggregation. Page level filters provide for this capability. Filter on the page, which will give you a summary portlet along with detailed rows with no summarization.

- Set grid portlets to NOT auto display results. If there is a business requirement that requires data to be shown when the page is first accessed, than a default filter should be set to limit the result set.

- Portlets should be avoided on the home page, or designed properly to allow quick end user access. A poorly designed filter (or no filter at all) may launch a query returning all rows in the database while the end user is trying to login. This time needed to execute a poorly filtered or poorly optimized query can prevent the end user from accessing their home page and look like a "system down" condition, resulting in unnecessary escalations.

- Minimize the amount of columns to return via your portlet. Clarity has the ability to return as many columns as you want but there can be a performance impact

for doing this. Do not use portlets as an "ad-hoc" vehicle, (selecting many columns into the query that are not displayed, in case the user wants to add columns to their grid later. This type of activity is should be managed in Business Objects. You should limit the amount of columns returned to 10-15 at the most.

- Restrict the use of the "configure" option on portlets on users' home pages to further restrict the number of columns that can be displayed.

- Label portlet fields not in use should with a "z" to group them in the bottom of the field list to prevent users from adding them as columns to sort on.

4.3 GEL Scripting

GEL is a powerful language that gives you direct access to the Clarity schema from within the constructs of the Clarity application. Reference the Clarity PPM Integration guide for further information on GEL.

All GEL scripts must store their URL, usernames and passwords outside of the database. These must be stored in a custom.properties file so environment refreshes don't impact the database itself.

This is accomplished using the following steps:

1. Create a custom.properties files that has a minimum of the following parameters:

```
xog.url=https://<servername>/niku/xog
xog.user=<xog username>
xog.pass=<xog password>
```

2. Add an entry to the BG services JVM args that provides the location of the custom.properties file. This removes any file location specific information from the database schema (Change the path as required. On Linux, you can utilize a UNC path to a shared file.) Re-deploy the BG service(s).

```
-Dcustom.properties=d:/niku/clarity/config/custom.properties
```

3. The GEL scripts themselves can utilize this sample code to call the parameters in from the custom.properties file:

```
<!-- Read the local env custom properties -->
<!-- this will read the file defined by the -Dcustom.properties java argument -->
<core:catch var="exception">
   <core:invokeStatic className="java.lang.System" method="getProperty" var="customPropertyFile">
      <core:arg value="custom.properties"/>
   </core:invokeStatic>
   <util:properties file="${customPropertyFile}" var="customProperties"/>
</core:catch>

<core:if test="${exception != null}">
   <gel:log level="ERROR">Unable to open custom properties file</gel:log>
   <gel:log level="ERROR">Caught Exception was:
   ${exception}</gel:log>
</core:if>
```

4. Once loaded, you can reference the property file values as follows (java.util.Properties object instance), assuming there are values in the custom.properties file for xog.url, xog.user and xog.pass:

```
<gel:log level="INFO">URL = ${customProperties.get("xog.url")}</gel:log>
<gel:log level="INFO">URL = ${customProperties.get("xog.user")}</gel:log>
<gel:log level="INFO">URL = ${customProperties.get("xog.pass")}</gel:log>
```

All GEL scripts should follow these basic standards:

- All SQL and SOAP tags should be wrapped in gel exception handling.
- All SQL tags should utilize bind parameters.
- All debug statements should be controlled via a parameter that is optional when needed.
- All complex SQL statements need to be analyzed for performance implications. This can be done via EXPLAIN PLAN in a SQL tool.
- If the GEL script is performing XOG read/write actions in a loop, the XOG login and logout actions must be outside of the loop.
- SQL update/insert/delete statements should not be utilized.

4.4 Business Objects Reporting

Clarity utilizes SAP Business Objects for reporting.

When creating reports, the following best practices should apply:

- Crystal Reports should be designed to pull data via direct SQL or a SQL Stored procedure. Avoid using a universe for Crystal Reports. This allows the reports to be self contained or only have one depended objects, the stored procedure. Once a Crystal Reports is built off a Universe it is tied to that Universe by the CUID and cannot be repointed to a different universe.
- Do not use the Legacy CA_Clarity.unv Universe for creating reports, Crystal or WEBI. This universe exists only to support use of Legacy reports.
- Be sure to use SQL Comments in Reports SQL statements and Stored procedures indicating the Report and Author.

- Thoroughly test reports in an environment with data volumes equal or greater than production.
- Ensure that required filter parameters are used when deploying reports as jobs in Clarity so that a report cannot be run with wide open parameters and affect system wide performance.

4.5 Process Design

Clarity processes are workflow items. When creating processes, the following best practices should be applied:

- Create multiple generic XOG users when xogging via process. These XOG users would be restricted to just the rights required for the their particular area (i.e. XOGPROJ, XOGRESOURCE, etc)
- Try to keep start condition, pre-condition and post condition expressions as simple as possible.
- For a process with an on-update start condition, try to ensure that the actions the process takes result in the start condition evaluating to FALSE by the time the process finishes executing. This will prevent the process from firing over and over again.
- The number of active processes tied to an object definition does have an effect on performance due to the number of start condition evaluations that must be conducted upon UPDATE or CREATE. Try to keep the total number of active processes for a given object to as few as possible. For instance, have a single "create" process with appropriate branching logic instead of having multiple "create" processes. Some process instances may need to be kept around after they complete for auditing purposes. Identify processes that do *not* need to be kept around and schedule the Delete Process Instance job to delete them when they have completed (nightly or weekly basis). Frequently run processes are good candidates to delete as they typically carry almost no auditing benefit.
- Break up big processes into a series of smaller manageable sub processes. Processes are then easier to manage and perform better. Sub processes are a new feature in the Clarity 8.x release.

- Reduce the number of steps used by a process by performing more than one action within a step. This will create a more efficient and better performing process.
- Processes can span multiple objects. Instead of creating two or more separate processes, it might be more efficient to create one multi-object process. A multi-object process will involve multiple objects (ex: Ideas and Projects) allowing users to configure conditions and action items with information from the objects involved in the process. Users can configure links between objects via lookups and the process engine will recognize those links when creating a multi-object process.
- During BG services startup, the process engine loads, compiles, and caches all active processes. This can cause an initial spike in database CPU usage.

4.6 Implementing Auditing

The global audit trail provides a log of all audit records for all objects that are configured for auditing. This log is a record of additions, deletions, and changes for audited objects.

The records for an instance of an audited object persist in the global audit trail, even after the instance is deleted. In addition, a record is kept of the individual who performed the deletion of the instance.

When implementing auditing, consider the following best practices:

- Do not audit every attribute on an object. Carefully consider exactly which attributes must be audited.
- Specify a reasonable value for Purge Audit Trail on the object you are auditing. This is the maximum number of days the audit records will be kept before the Purge Audit Trail job removes them. Do not leave the default (empty) which is infinite retention.
- Be sure your Clarity instance has patches for defects CLRT-54361, CLRT-59682 and CLRT-54144. These defects are related to performance of both the Audit Trail portlets on object instances and the Administration Audit Trail page as well as the performance of the Audit Trail Purge job itself.

- Do not audit attributes that change very frequently (for instance, do not audit an attribute that is updated nightly by a scheduled job). The audit results will not be meaningful and will take up a large amount of space in the database.

4.7 Changes to the base calendar

Making changes to the base calendar will result in the reslicing of the following slices:

- Resource Slices (prj_resources.pravailcurve)
- Team slices (prteam.pralloccurve)
- Team hard curves (prteam.hard_curve)

This reslicing can potentially result in millions of records being resliced. Such action can cause the delay of other jobs and may also affect database performance. Changes to base calendars should be done in off hours and preferably on weekends.

The resulting reslice of changing a base calendar will also cause a longer run of the Datamart extraction job. Timings of the reslice and long Datamart run should be tested so expectations can be set when the change is done in production.

4.8 Rate Matrix Extraction

The Rate Matrix Extraction (RME) Job generates and updates rates and costs within the Clarity application for use in different areas of the application.
NBI_PROJ_RES_RATES_AND_COSTS is the final table where the data resides at the end of the RME job.

A **full RME** run is accomplished by selecting the **first 3 options** as below and the duration of the run will depend on the number of investments, resources, teams, assignments and tasks. The run also depends on the number of matrices available on the system. Best practice is to run this once per week or during the off peak hours.

1. Extract Cost and Rate Information for the Scheduler
2. Prepare Rate Matrix Data
3. Update Rate Matrix Data

An **Incremental RME** run is accomplished by selecting **all 4 options** as below and the duration depends on the number of investments changed from the prior full run.

1. Extract Cost and Rate Information for the Scheduler
2. Prepare Rate Matrix Data
3. Update Rate Matrix Data
4. Incremental Update Only

The following 3 jobs will update the projects' *last_updated_date* field. This will cause the incremental RME run to take longer than expected.

- LDAP sync user Job
- Investment Allocation Job
- Earned Value update Job

The workaround is to schedule these jobs prior to a full RME run followed by an incremental RME runs.

For Oracle, this job will utilize the number of CPU processes available on the database server, and utilize them as it sees fit to automatically parallelize the job.

4.9 Timeslicing and Datamart

Clarity stores time-scaled data in single data entry in the database as a Binary Large Object field (BLOB.) In some cases more than one type of data is stored in a BLOB. BLOBs cannot be read by reporting tools, therefore it's necessary to flatten out this data into a readable format. A time slice is a flat table that contains data that is derived from a sliced binary large object (BLOB).

Timeslicing is performed by the Timeslicing background job. The Timeslicing Job extracts the BLOB data into a readable flat table based upon the criteria you set in the time slice request. The flattened timesliced data can be queried for reports and portlets or used in data extracts.

The Datamart is a back ground job that processes timeslice data into aggregated tables and summarizes investment data.

Changes to the timeslice configuration should be made off hours and preferably on a weekend as they may take some time to re-slice and generate a significant amount of database activity.

Daily slice data is essentially replicated for use in the Datamart. We recommend keeping the daily timeslices to a minimum, and using monthly or weekly slices when possible. Given that the Datamart relies on the daily slices we recommend that all new custom reports be written to pull data directly from timeslices at the monthly or weekly level.

The 5 standard daily timeslices are used in the Datamart:

ID	**Request Name**
1	DAILYRESOURCEAVAILCURVE
2	DAILYRESOURCEACTCURVE
3	DAILYRESOURCEESTCURVE
10	DAILYRESOURCEALLOCCURVE
11	DAILYRESOURCEBASECURVE

When configuring timeslices, the following best practices should be applied:

- Keep Daily timeslices to a minimum
- Stay within 3 to 6 months in the past and 3 to 6 months in the future.

The daily timeslice configuration below will cover a moving 6 month window with 3 months in the past the current month and 3 months in the future.

- **Actuals**: If you require daily actual data for 3 months, we recommend that you set your From Date for the Actuals slice request to the start of a month 4 months in the past with the Number of Periods to 124, so that it overlaps the current month. If you need to look further back we suggest using monthly data.
- **Estimate to Complete:** There is no need to slice estimates in the past. We recommend that you set the "From Date" to the start of the current month and number of periods to 124. If you need to look further we suggest using monthly data.
- **Baselines:** If your organization doesn't maintain project baselines set the number of periods to Zero. The logical configuration for Baseline slice data should start at the beginning of actuals and extend through estimates. Set the From Date to same From Date as actuals slice request and change the number of periods to 214.
- **Availability:** The configuration for Availability slice data should start at the beginning of actuals and extend through estimates and baselines. Set the From Date to same from date as actuals slice request and change the number 214.
- **Allocations:** If you are not maintaining allocation at the Project level company wide, you may have no need to maintain slice data for allocation. This is by far the largest portion of slice data, and if it is not entirely needed it can be dramatically reduced. We recommend that if you do not set project level allocation, you should set the "Number of Periods" to "0" for the Allocation slice request. This will minimize the amount of data that is being stored for Allocation slices.

5 Customizations

In general, customizations should be avoided whenever possible. Sometimes we realize they are necessary, so offer the following guidance based upon our internal coding standards.

Specific guidelines should to be followed for the various types of customizations that can be done to Clarity:

- Change Control
- Java code
- Database Objects
- Scheduled Interfaces
- Changes to Clarity files

5.1 Change Control

All customization to Clarity must follow a rigorous change control process to ensure that each change to the system is recorded, reviewed internally, understood, and tested. This is especially critical when it comes to migrating customizations from one release to the next or from one system to another. It is the customer's responsibility to ensure such procedures are in place.

5.2 Java Code

All java code that is compiled and running within a server must to follow these basic standards:

- Reference the GEL Section of this document how to properly set and invoke URL's, usernames and passwords. Example:

```
try {

  String propFile = System.getProperty("custom.properties");

  if (propFile != null) {
    FileInputStream f = new FileInputStream(propFile);
    Properties customProperties = new Properties();
    customProperties.load(f);
```

```
          f.close();

          System.out.println("XOG URL = " + customProperties.get("req.xog.url"));
        }
        else {
          System.out.println("Unable to determine custom property file path.  Ensure -
Dcustom.properties is set in java arguments");
        }


      } catch (Exception e) {
        e.printStackTrace();
      }
```

- All SQL must utilize bind variables.
- Exception handling must be complete (finally blocks to closw SQL stmt/conn).
- Any XOG loops performance a single login/logout.
- All complex SQL statements need to be analyzed for performance implications. This can be accomplished with execution plans.

5.3 Database Objects

Custom database objects consist of the following:

- Indexes
- Stored Procedures
- Tables
- Views
- Functions
- Triggers

All custom database objects conform to a universal naming standard. This is normally a "Z_" prefix but can be any other uniquely identified character combination (i.e. "gse_" or "gd_", etc.)

A detailed schema definition or DDL is required that documents all custom database objects and their intended purpose. There should be clear comments in the objects

themselves (where pertinent) so the database code can be quickly reviewed and understood.

5.4 Scheduled Interfaces

A scheduled interface usually consists of a package or a combination of custom database objects, java code and/or GEL scripts. Data most commonly imports into the system via some kind of scheduled job. Clarity should not be utilized as a "middleware" system. Where possible, all data should be acted on via XOG from properly formatted and pre-validated XML files.

When creating interfaces, the following best practices must be followed:

- All interfaces must be self governing. They must be configured to manage the expected volume (with a high ceiling trigger) and either gracefully fail or only process the amount of the data to the high ceiling trigger.
- All interfaces must be made resilient and hardened to deal with mal-formed or duplicate incoming data. The interfaces should fail gracefully if these conditions are met.
- When creating custom jobs you must consider the other jobs and the schedule they run on. Interface jobs may need to be altered to be incompatible with other Clarity stock or custom jobs if performance problems can occur. All custom jobs that interact with custom interfaces must have test cases that map out expected run times with the expected data load the will be importing/exporting.
- All data being imported through XOG must be limited to batches of 200 records per execution.