Best Practices - PDI Design Guidelines

This page intentionally left blank.

Contents

Contents, cont'd

Overview

This document is intended to provide best practices for designing and building your Pentaho Data Integration (PDI) transformations and jobs. This will result in maximum speed, reuse, portability, maintainability, debugging and knowledge transfer. The document is arranged in a series of topic groups with individual best practices for the topic explained. It is not intended to demonstrate how to implement each best practice or provide templates based on the best practices defined within the document.

Software	Version
Pentaho	5.4, 6.x, 7.0

Server Configuration

This section contains steps for enabling Spoon connections, naming those connections, and tips for avoiding JNDI. You will also find guides for repository and database usage.

Enable Spoon Connection Option

- **Recommendation:** Enable the Spoon ONLY SAVE USED CONNECTIONS TO XML option.
- **Rationale:** Spoon will write all shared connection information into each transformation or job file, by default.
- **Solution:** This keeps only those connections used by that specific transformation or job in the XML file.

Avoid Using JNDI

- **Recommendation:** Do not use JNDI or similar settings used in enterprise applications for PDI.
- **Rationale:** PDI has its own algorithms for allocating and using unique connections per step instead of JNDI and connection pooling.
- **Solution:** Avoid using JNDI in PDI unless the `.ktr` will run inside the BA Server. Use variables to parameterize/hide the connection credentials. Avoid connection pooling, as well.

Name the Connections

- **Recommendation:** When naming your connections, avoid using the words Production, Development, Oracle, and MySQL.
- **Rationale:** Variables make changes to databases over time. You would want to avoid having to change all instances of MySQL when you migrate to Oracle. When migrating from Production to Development, change the underlying `KETTLE_HOME`. The changes will apply in that new environment.
- **Solution:** You can swap out your connections by changing the `KETTLE_HOME` variable.

Use a File Repository

- **Recommendation:** Always use one of the provided repository types rather than using direct system files.
- **Rationale:** By using a file repository, you can still add, modify, or delete the files but they are READY to be uplifted into an Enterprise Repository in the future. When calling jobs and transformations in PDI, you need to choose a file or repository when developing. Always use the repository option, but you can change between files and Enterprise Repository.
- **Solution:** Use a file repository to organize your files, if you are not using the Enterprise Repository.

Use SQL for the Database

- **Recommendation:** Do not use PDI steps for features that a database could perform.
- **Rationale:** A database is often more efficient at sorting and joining the Pentaho. This is not absolute and can lead to overly complex SQL input if abused.
- **Solution:** Use the SQL for the database of your DBMS platform.

Avoid Stored Procedures and Database Views

- **Recommendation:** Avoid using database-specific features.
- **Rationale:** Views are typically slow to return data. It will be more efficient to replace the view SQL in the PDI input step than to call that view with an additional `WHERE` clause. Stored procedures are okay for inputs, but should not be used for lookups, as they are much slower than PDI. PDI can process several thousand records per second. If each of those rows must go out to a database and run a stored procedure, it will be slower.
- **Solution:** Avoid stored procedures for lookups and database views for inputs.

Development for Transformations

In this section, you will find step-by-step best practices for transformation development. You will be led through pre-job entry transformations, describing transformations, and how to avoid overloading those transformations.

Pre-Job Entry Transformations

- **Recommendation:** Design and build from the inside out.
- **Rationale:** This process allows you to design, code, debug, and test each individual piece without requiring you to run it as part of the larger process. This is key to a modular design.
- **Solution:** Create a transformation before creating the job entry that calls the transformation. Create sub-jobs before creating the jobs that call them.

Describe Transformations and Jobs

- **Recommendation:** Assign at least one descriptive note to each transformation and job.
- **Rationale:** Adding notes allows those reviewing the code or taking over support to understand decisions behind the logic or changes.
- **Solution:** Major changes to the logic or flow should also specify who made the change.

Avoid Overloading Transformations

- **Recommendation:** Do not put processing of multiple source systems into one transformation.
- **Rationale:** Data processes can get mixed up if they are together. Additionally, if one source system is down, you need to be able to process others without impacting processing. This is harder if they are wired together in the same transformation and job-flow.
- **Solution:** Split processing into one transformation per source system for the type of destination data.

Create Transformations One Step at a Time

- **Recommendation:** Create your transformations one step at a time.
- **Rationale:** This makes testing and debugging easier, as you are adding one new variable, process, or step to the transformation at a time.
- **Solution:** Create your transformations starting with the input. Test existing steps before adding another step. Add more steps only after receiving the expected result. Do not add or try to wire up multiple steps at the same time. This allows you to start from a working process each time you add new steps to a transformation.

Name Consistently

- **Recommendation:** Name transformations, jobs, and steps consistently using the same conventions.
- **Rationale:** This naming convention will allow you to see what type of task is being performed when reviewing logs, files, and database logging.
- **Solution:** Name each job entry with a prefix of the type, followed by descriptive text. Name each transformation step with a prefix of the type of step, followed by descriptive text.

Parameterize Jobs and Transformations

- **Recommendation:** Do not parameterize before the basic transformation is working without variables. Parameterized jobs and transformations only work once with fixed values.
- **Rationale:** This allows you to start from a working process with less complexity. It also makes testing and debugging easier, as you are parameterizing what was once already working. Trying to debug parameters, and the logic of the transformation, is difficult.
- **Solution:** Create your transformations starting with fixed inputs, outputs, and configurations. Introduce variables before deploying them into the server.

Label with Dummy Steps

- **Recommendation:** Use dummy steps to label data flows.
- **Rationale:** This approach allows you to see the number of rows flowing through this branch in the logs. It also buffers you from downstream changes. Only the steps after the dummy step need to change, and do not impact the branching.
- **Solution:** Rename each dummy step to better describe the data that flows through that part of the transformation. Use this technique after each filter, case, error, or any other type of branching in the data flow.

Controlled Environment Transformations

- **Recommendation:** Develop in a controlled desktop environment.
- **Rationale:** Prior to deployment, maximize your time spent in Spoon. This makes transformations easier to build, test, and debug. Migrate from there to Kitchen or DI Server only after things are working and fully parameterized.
- **Solution:** Create and unit-test your transformations using Spoon, while disconnected from a server. Run the job via Kitchen, once the process is tested and working properly. Migrate the code to a server environment running the DI Server or Carte, if successful.

Using Variables

This section provides information on parameters for variables, KETTLE_HOME variables, and variable use for external references.

Use Parameters for Variables

- **Recommendation:** Use parameters for variable definition instead of variables.
- **Rationale:** This approach allows you to test each transformation and job without relying on global variables. At runtime, you can set a parameter to different values to influence the behavior.
- **Solution:** Parameters should be used to pass variables into jobs and transformations. Variables should only be used to set global variables for an entire job flow.

Separate KETTLE_HOME Variables

- **Recommendation:** Do not mix `KETTLE_HOME` files for multiple projects, customers, or environments such as the Development, QA, and Production servers.
- **Rationale:** When `KETTLE_HOME` files are shared between projects, values can be mixed and shared improperly. This can be as much a security concern as a data issue.
- **Solution:** Use a different `KETTLE_HOME` variable for each of these environments, customers, and projects. Pentaho scripts know how to use the `KETTLE_HOME` variable at startup to alter the files used for that runtime environment.

Use Variables for External References

- **Recommendation:** Use variables for any external reference outside of Pentaho.
- **Rationale:** Using this approach makes it easier to migrate from Development to QA and Production, etc. It also allows you to externalize security-sensitive connection information so that developers do not need to know the passwords in order to use the connection.
- **Solution:** All external references should be replaced with variables and/or parameters. This applies to host, user, password, directory, filename, etc.

Validate Job Variables

- **Recommendation:** Validate any variables or settings required for proper operation, prior to starting the main job.
- **Rationale:** This avoids a job getting deeper into execution only to find that the environment was not setup properly at the start. This is especially important during transitions between `DEV`, `TEST`, and `PROD`.
- **Solution:** Validate all external variables and their structures, to avoid later connection errors. Data validation steps can be used inside transformations, as well as job-level connections and table-testing.

Logging

Here, you will find ways to navigate through logging operations such as redirecting output, tracking audits, and handling errors upon root job fails. You will also find steps for Kettle logging, row-level logging, and more.

Use Logging Tables

- **Recommendation:** Always use `JOB`, `TRANSFORMATION`, and `CHANNEL` logging tables.
- **Rationale:** This auditing allows you to track performance over time.
- **Solution:** Use the `kettle.properties` variables instead of creating your own variables or selectively defining which transformations and jobs get logged.

Redirect Output to Kettle Logging

- **Recommendation:** Redirect all output to KETTLE logging destinations.
- **Rationale:** Without turning these variables to Y (N by default), `STDERR` will receive additional information useful for logging and debugging errors.
- **Solution:** Use the `kettle.properties` variables of `KETTLE_REDIRECT_STDERR` and `STDOUT=Y`.

Sub-job for One Log File

- **Recommendation:** When executing jobs in DI Server or Carte, use a sub-job that writes to one log-file for the entire execution.
- **Rationale:** This organizes all job-related logging into one file. Research is eased, when that server runs more than one job at a time, while evaluating an error in a job. All logs are placed into one log file for the server, without this.
- **Solution:** Complete the logging tab on the job entry of the root-job, when launching the sub-job.

Track Audits on Target Tables

- **Recommendation:** Place audit fields on each target table.
- **Rationale:** You will be allowed to cancel certain records or an entire batch, after the fact. The root channel ID can be captured for the root job. It is used on the channel log table to keep track of all transformations and jobs that start under that root job.
- **Solution:** Place the batch, job, or root-channel ID on each target table to keep track of which records were loaded by which job.

Function for Row-Level Logging

- **Recommendation:** Use the JavaScript `writeToLog()` function for more formatting precision on row-level logging.
- **Rationale:** Row-level logging is useful to track the branching and flow of data during development and testing. Be sure row-level logging is disabled after testing. This function will be disabled, by default, as most production environments do not use detailed logging or higher.
- **Solution:** Set the level to DEBUG, when using this approach. Levels below should not be used for row-level logging of transformations. The highest row-level already logs rows, so this is only useful at DEBUG or DETAILED. You can use `writeToLog()` if you have single-row transformations where each field has its own row.

Error Handling for Root Job Fails

- **Recommendation:** Enable error-handling for all failure cases of all root job entries, as well as for transformation steps that can produce an error or exception.
- **Rationale:** Rows can be handled individually when row-level error-handling is used. The transformation may continue successfully without processing those rows in error. They can still be processed in subsequent transformations. All job-level failures need to be logged or acted on by operators.
- **Solution:** Each job entry has a TRUE and FALSE path. The FALSE path should always point to a failed action. All sub-jobs and/or sub-transformations will trigger the fail-proof root-level handlers set in place. For transformation steps, right-click on each and chose ENABLE ERROR HANDLING. Make sure to choose the right step (likely a dummy step) and add on all the related error-handling fields for context.

Mondrian Cache

In this section, you will find information on clearing and priming the Mondrian cache, as well as JVM job execution.

Clear the Mondrian Cache

- **Recommendation:** Clear the Mondrian cache after each ETL load.
- **Rationale:** Data used in Analyzer can be out of sync with the database if new data is loaded while the old data has been cached. This process will trigger the application to re-query the database to get the latest data after the load.
- **Solution:** Clear the Mondrian cache in PUC or use an `HTTP` step in PDI to automatically clear the cache.

Prime the Mondrian Cache

- **Recommendation:** Prime the Mondrian cache after clearing it.
- **Rationale:** Without priming the cache, users will experience longer load times with an empty cache.
- **Solution:** Run `xaction` scripts, CDA, or schedule Analyzer reports to run on a schedule to fill the cache.

Personal JVM Job Execution

- **Recommendation:** For loads that do not happen continuously, execute jobs in their own Java Virtual Machine (JVM) using Kitchen.
- **Rationale:** Shorter running JVMs are less susceptible to memory issues than longer running JVMs.
- **Solution:** Use Kitchen to execute infrequently running jobs, and to log their output separately.

JSON Parsing

This section provides steps for separating tasks with JSON parsing, how to use JavaScript for many levels of JSON, and ways to expedite parsing.

Separate Tasks with JSON Parsing

- **Recommendation:** Process as one object per row rather than one object per file.
- **Rationale:** When parsing a JSON file, the input step must read the entire file before it begins parsing. This will limit the size of the file that can be useful.
- **Solution:** Make the JSON input data split at one valid object per row. Use the text file input step to read the raw data, and the JSON input to parse each row.

Use JavaScript for Multi-level JSON

- **Recommendation:** Use JavaScript over JSON when parsing multi-level JSON.
- **Rationale:** The JavaScript step is more efficient at multi-level parsing and can be easily configured to parse many levels in one pass.
- **Solution:** If your JSON has embedded arrays in objects, use the JavaScript step to parse the entire object rather than multiple JSON input steps.

Expedite Parsing

- **Recommendation:** Use multiple copies of JSON/JavaScript steps to speed up JSON parsing.
- **Rationale:** The JSON step can only pass one level at a time. JSON parsing is CPU intensive. If you can split up the task across multiple cores, it will be faster. This is only possible if you are reading each row as an object.
- **Solution:** Enable multiple copies by right-clicking on a step and choosing CHANGE NUMBER OF COPIES TO START.