HITACHI
Inspire the Next

Pentaho Data Integration (PDI) Project Setup and Lifecycle Management

This page intentionally left blank.

Contents

Overview

Starting your Data Integration (DI) project means planning beyond the data transformation and mapping rules to fulfill your project's functional requirements. A successful DI project proactively incorporates design elements for a DI solution that not only integrates and transforms your data in the correct way but does so in a controlled manner.

You should consider such nonfunctional requirements (NFRs) before starting the actual development work to make sure your developers can work in the most efficient way. Forcing such requirements in at the end of the project could be disruptive and hard to manage.

Although every project has its nuances and may have many nonfunctional requirements, this document focuses on the essential and general **project setup and lifecycle management requirements** and how to implement such requirements with Pentaho Data Integration (PDI).

The document describes multiple DI solution design best practices. It also presents a **DI framework** solution that illustrates how to implement these concepts and separate them from your actual DI solution. This framework is nothing more than a set of jobs and transformations that acts as a wrapper around your actual DI solution, taking care of all setup, governance, and control.

Consider the following common project setup and governance **challenges** that many projects face:

- Multiple developers will be collaborating, and such collaboration requires **development standards** and a **shared repository** of artifacts.
- Projects can contain many solutions and there will be the need to **share artifacts** across projects and solutions.
- The solution needs to integrate with your **Version Control System** (VCS).
- The solution needs to be environment-agnostic, requiring the **separation of content and configuration**.
- **Deployment** of artifacts across environments needs to be automated.
- The end result will be supported by a different team, so **logging and monitoring** should be put in place that supports the solution.
- Failed jobs should require a simple restart, so **restartability** must be part of job design.

The intention of this document is to speak about topics generally; however, these are the specific versions covered here:

Software	Version(s)
Pentaho	7.x, 8.x, 9.0

The Components Reference in Pentaho Documentation has a complete list of supported software and hardware.

Before You Begin

This document assumes that you have knowledge about Pentaho Data Integration (PDI) and that you have already installed the Pentaho software to be able to run the demo that comes with this best practice document to illustrate the concepts covered.

Terms You Should Know

Here are some terms you should be familiar with:

- **Local (local) environment**: developer's local machine used for ETL development and unit tests through Spoon
- **Development (dev) environment**: Pentaho Server environment used for system tests
- **Test (test) environment**: Pentaho Server environment used for user acceptance tests
- **Production (prod) environment**: Pentaho Server environment used as production environment

Use Case: Sales Reporting and Data Exports

Throughout this document and the demo, we apply the concepts covered to the following sample use case:

Janice works for an online retailer and is working on two DI projects:

- *Creating a sales reporting data warehouse*
- *Generating data exports for a tax agency*

The projects have the following environments available, each with a dedicated function. Together, these environments define the lifecycle of the solution before it hits production:

- *Local*
- *Dev*
- *Test*
- *Prod*

Demo Download Link

The demo that comes with this best practices document to illustrate the concepts covered is available on HCP Anywhere.

Content and Configuration Management

Before exploring the DI framework solution, illustrating concept details and showing how to implement them with PDI, we will introduce the following **development and lifecycle management concepts**:

- Managing Your Content
- Managing Your Configuration

*All **jobs and transformations** discussed in this document are available within the demo for inspection and usage.*

Managing Your Content

Use version control when you work on a DI project with many contributors. Store your DI project content (PDI jobs and transformations, external SQL or other scripts, documentation, or master input files) in a central, version-controlled repository. This repository should also support other team collaboration and release management capabilities.

Although the Pentaho repository offers a basic revision control system, it is not intended to replace common market tools, such as Git. If you need the advanced features of an enterprise VCS, we recommend that you use one.

Version control for the Repository is disabled by default. The version history in this case is handled by Git. There is no need for special configuration on the developer machines, because they will not be using the Pentaho Server. Use Version History has more information about configuring this.

It is recommended to integrate PDI with Git repositories for storage and version control of the project's DI artifacts throughout the development lifecycle. However, the project does make use of the Pentaho Repository as a container for the jobs and transformations, once the Git repositories deploy a release. This will be true for all environments where the Pentaho Server is involved (development, test, and production).

This requires you to design your solution so that it can work well in both environments:

- Working and executing file-based during development to integrate with Git.
- Executing on the Pentaho Server, using the Pentaho Repository to store jobs and transformations.

Although the examples in this document make use of Git, this document contains project setup information that will be relevant for you even if you do not use Git. For those using a VCS other than Git, note that these instructions may be inappropriate for the alternative VCS. For advice on VCS-specific integrations, please contact Hitachi Vantara.

Jobs and transformations will be deployed to the Pentaho Repository only on those environments that involve the Pentaho Server and use the Pentaho Repository. Other solution artifacts such as SQL

scripts and shell scripts do not deploy to the Pentaho Repository, instead remaining on the filesystem. The following sections detail these operations.

Configuration Primer

The configuration repository holds all the needed configuration properties to develop and run the ETL, separated by project environment. This forces the developer to parameterize the ETL. As a result, the solution becomes environment-agnostic, a key attribute powering the solution's lifecycle management. Our project's configuration repository has one folder per project environment:

Table 1: Configuration Repository Structure

Folder	Contains Configuration Details For:
`config-local`	Developer's local machines
`config-dev`	Development environment
`config-test`	Test environment
`config-prod`	Production environment

Use the following subfolders to structure the content stored in your `config` folders. All `config` folders have these folders available, and they play a key role in your DI framework:

Table 2: Subfolders for Config Folders

Folder	Description
`.kettle`	The `KETTLE_HOME` folder used by PDI to hold its default application and configuration files
`metastore`	Run configuration and slave server details
`properties`	Specific project properties files where project specific variables get introduced

Git Repositories

Our DI project runs in different environments as it moves through the development and test lifecycle. This means that you need to separate the DI artifacts and code from the configurations that make the code run properly.

The ETL must be designed in such a way that all external references (such as database connections, file references, and all internal calls to jobs and transformations) are made using variables and parameters, which get their values from the config files part of the configuration repository.

The concepts and demo discussed in this document focus on two DI projects, creating a sales data warehouse (`sales_dwh`) and generating data exports (`data_export`). Since Git's main functionalities like branching and merging work at the level of the repository, and because we want to have maximum flexibility and power for our two projects, we split them up into separate Git repositories.

Additionally, to enforce the **separation of content and configuration**, we end up with the following four main Git repositories:

Table 3: Git Repositories

Git Repository	Description
`sales_dwh`	This repository holds all the `sales_dwh` project's ETL code and other DI content artifacts. *In this document, we will refer to such a repository as the **ETL repository**.*
`sales_dwh-configuration`	This repository holds all the necessary configurations, separated by environment, necessary to run the ETL for the `sales_dwh` project. *In this document, we will refer to such a repository as the **configuration repository**.*
`data_export`	This repository holds all the `data_export` project's ETL code and other DI content artifacts. *In this document, we will refer to such a repository as the **ETL repository**.*  *Note that the demo does not include this repository. We include it in this table to clarify the concept of project and project-configuration repositories.*
`data_export-configuration`	This repository holds all the necessary configurations, separated by environment, necessary to run the ETL for the data_export project. *In this document, we will refer to such a repository as the **configuration repository**.*  *Note that the demo does not include this repository. We include it in this table to clarify the concept of project and project-configuration repositories.*

All project environments will have these repositories checked out from Git onto the local filesystem of the server.

PDI-Git Integration

Every developer checks out the needed Git repositories onto their local filesystem when developing, working with PDI in a file-based method without a Pentaho repository. The same way of working holds true for most version control systems. Working file-based has consequences for things like [sharing database connections].

ETL Repository Structure

The ETL repository holds all the project's ETL code and other DI content artifacts. Use the following folders to **structure** the content stored in the project's `etl` repository. All projects' `etl` repositories have these folders available, and they play a key role in your DI framework:

Table 4: ETL Repository Folders

Folder	Description
`content-pdi`	Hosts all the PDI jobs and transformations (subfolders will be created to structure the ETL even more)
`file-mgmt`	Hosts all input and output files related to the project
`scripts`	All the script files the solution uses
`sql`	The necessary data definition languages (DDLs) to create the necessary database tables
`log`	The project log folder to where all project log execution should be pointed. Add a `.gitignore` rule to the repository to make sure the content of this folder never hits the central repository on Git. Later in this document, you will see that you can also configure logging to be central for the server and not solution specific.
`documentation`	Markdown notation for relevant documentation to address the main logic decisions under the current sub-project

If needed, you can split these mandatory folders further. For example, for the `sales_dwh` project, your ETL jobs and transformations follow a typical data warehouse flow going through staging, operational data store (ODS), and warehouse. Because of this, you should group your content in the `content-pdi` folder into three subfolders: `staging`, `ods`, and `warehouse`.

This creates the following structure for your Git `etl` repositories:

```
|-- sales_dwh
| |-- content-pdi
| | |-- staging
| | |-- ods
| | |-- warehouse
| |-- file-mgmt
| |-- scripts
| |-- sql
| |-- log
| |-- documentation
|-- data_export
| |-- content-pdi
| |-- file-mgmt
| |-- scripts
| |-- sql
| |-- log
| |-- documentation
```

Git Repository Branches and Tags

These branching and tagging examples only show ways that you can use certain Git features to support the lifecycle management of your project's `etl` and `configuration` repositories. They are not necessarily requirements.

Make sure to consult your company or industry standards and adapt accordingly.

Git configuration can be done through a Git client.

Git Branches

Git repository branches provide an isolated environment for applying new features to your DI solution. This helps you thoroughly test the features before integrating them into your production-ready code. Isolation improves collaboration since multiple features can be implemented in parallel without conflicting with each other.

Branching does require a configuration manager, so consider whether you want to add this feature based on your team's size and progress in Git adoption.

For each Git repository in your project (`etl`, `configuration`), create the separation between development and production code with at least the following permanent branches:

Table 5: Git Repository Branches

Branch	Description
`Dev`	This branch holds the ETL code and the configurations that are under development.
`Master`	This branch holds the ETL code and the configurations that are ready for production deployment.

Updates between the two branches (either by merge or by pull request) should always be made through a formal request and performed by a responsible party.

Both repositories could also be controlled by **limited lifecycle branches**. Create these branches to meet development needs. Their names should match their purpose. Here are a couple examples of limited lifecycle branches:

Table 6: Git Repository Limited Lifecycle Branches

Branch	Description
`sprint-X`	Branch created to support the development for sprint X and shared by all developers on the sprint.
`sprint-X-feature-Z`	Branch for feature Z that is being created during sprint X to allow some developers to work on a specific feature.

Merge limited lifetime branches with the dev branch after the unit and system tests are performed and accepted, or with the master branch when the user acceptance tests (UAT) are performed and

accepted. The `merge` between these two branches should be formally requested with a pull request. The QA team then tests and accepts the merge request.

Git Tags

In addition to branches, the project can also make use of tags, which are human-readable labels that you can add to your commits. This way, Git can tag specific points in history as being important.

Tags should have names that are unique in the entire Git database. You can use this functionality to mark system tests, UAT, or production releases.

An example is release tags, on master/production branches, to quickly go back if something goes wrong with a new production feature. The same can apply for UAT, SIT, and so on, so that the test team can revert to a point in time before the problem occurred, and resume testing.

Deployment Strategy

The deployment process makes sure all necessary project artifacts become available on the required Pentaho Server machines as your project moves through its lifecycle.

The Git repositories are checked out to the local filesystem of the Pentaho Server machines after you deploy a test or production release. Only the jobs and transformations from the `etl` repository end up in the Pentaho Repository. All other artifacts stay on the filesystem so that they can be referenced from the jobs and transformations at execution time.

> *For example, a job might reference a shell script to assist the job in executing a specific task. This script will be available from the filesystem and not from the Pentaho Repository.*

The PDI toolkit offers multiple features that can support and automate your deployment process.

Managing Your Configuration

Each folder in the project's configuration repository represents a different project environment configuration. The separation between **content** (`etl` repository) and **configuration** (`configuration` repository) enforces the creation of a neutral solution, a key attribute powering the solution's lifecycle management.

From the Content Management section, you already know that the configuration repository is divided into the following folders, matching the available environments on the development pipeline:

- `config-pdi-local`
- `config-pdi-dev`
- `config-pdi-test`
- `config-pdi-prod`

Each of these folders holds the following subfolders:

- `.kettle`
- `metastore`
- `properties`

Each environment folder also has an environment configuration file (`env.conf`) and multiple script files to make sure the PDI Java Virtual Machine (JVM) is started with the environment-specific configurations. These will be described later:

- `spoon.bat/sh`
- `pan.bat/sh`
- `kitchen.bat/sh`
- `start-pentaho.bat/sh`

The env.conf configuration file is there to configure the scripts on where to find the local Pentaho and Java installation:

Table 7: `env.conf` Configuration File Variables

Variable	Description	Example
`PENTAHO_HOME`	Local Pentaho installation directory	`/opt/pentaho`
`PENTAHO_JAVA_HOME`	Local Java installation directory	`/opt/pentaho/java`
`PENTAHO_DI_JAVA_OPTIONS`	JVM memory settings	`-Xms1024m -Xmx2048m`
`KETTLE_CLIENT_DIR`	*Optional*. Makes it possible to overwrite the PDI client installation directory. Defaults to `<PENTAHO_HOME>/design-tools/data-integration`	`/opt/client_tools/data-integration`

By separating the configuration from the actual code, the ETL solution can run in all environments without any changes to the ETL code.

The following sections provide more configuration details.

Default PDI Configuration in the .kettle Folder

The `.kettle` folder holds the default PDI configuration files. The following ones will be critical to your DI framework:

Table 8: `.kettle` Folder PDI Configuration Files

Folder	File	Description
`.kettle`	`kettle.properties`	Default environment variables available to PDI
	`shared.xml`	Shared connections for working file-based
	`repositories.xml`	Pentaho repository connection information
	`.spoonrc`	Spoon GUI options (such as grid size)
	`.languageChoice`	User language for Spoon

This folder is located by default under the user's `home` directory but can be loaded to a different directory to centralize this between all developers. Centralizing the `.kettle` folder under the configuration repository in Git (`configuration/<config-env>/.kettle`) guarantees that all developers share the same `.kettle` settings per environment.

You can change the default location of the `.kettle` directory by setting the `KETTLE_HOME` environment variable, pointing to the folder holding the `.kettle` folder.

In each environment, whenever you start PDI (client or server), you specify the correct **`KETTLE_HOME`** variable. This way, PDI starts up with the dedicated `.kettle` settings specific for that environment (local or server configuration properties).

Variables will be introduced from the outside to your DI framework and the actual DI solution in two main ways: `kettle.properties` (`.kettle` folder) and the `properties` files (`properties` folder).

Global Variables in kettle.properties

The **`kettle.properties`** file is the main configuration file for PDI. The DI framework uses this file to make certain key variables available to the framework and the actual DI solution. The variables in this file are considered global, shared among your individual projects.

The benefit of declaring global variables with the `kettle.properties` file is that they are, by default, available to your PDI JVM. The downside of `kettle.properties` when using the Pentaho Server is that you require a server restart to pick up new entries in this file.

Each configuration environment has its own version of this file, with the correct values for each of the key variables that are needed by the DI framework:

Table 9: Variables

Variable	Description	Example
`ROOT_DIR`	Filesystem path to the project's parent folder. This always points to the filesystem.  *This variable, included for completeness, is not part of the `kettle.properties` file and will be set in one of the later detailed scripts.*	`C:/Github`
`CONTENT_DIR`	The root folder to the ETL artifacts. Depending on the environment, it can have a filesystem path (developer machines), or a Pentaho Repository path (all other environments). Using both a `ROOT_DIR` and a `CONTENT_DIR` variable gives you the flexibility to have some artifacts on the filesystem and some in the Pentaho Repository.	• `local: C:/Github` • `server: /public`
`CONFIG_DIR`	Filesystem path to the configuration folder used by the current environment. The configuration repository will never be deployed to the Pentaho Repository, therefore this variable inherits from `${ROOT_DIR}`.	`${ROOT_DIR}/`

Variable	Description	Example
LOG_DIR	Filesystem path to the root directory that holds the logging folder, inheriting from `${ROOT_DIR}`. Later in this document, you will see that you can also configure logging to be central for the server and not solution specific.	`${ROOT_DIR}`
FILE_MGMT_DIR	Filesystem path to the root directory that holds the `file_mgmt` folder, inheriting from `${ROOT_DIR}`.	`${ROOT_DIR}`
PDI default logging variables	Please refer to the Logging and Monitoring section.	

DI_SERVER	DI Server connection details (`DI_SERVER.PROTOCOL`, `HOST`, `PORT`, `WEBAPP`)	`DI_SERVER.PROTOCOL=http`
		`DI_SERVER.HOST=localhost`
		`DI_SERVER.PORT=8080`
		`DI_SERVER.WEBAPP=pentaho`
		`DI_SERVER.USERNAME=admin`
		`DI_SERVER.PASSWORD=password`

As you can see, most of these global variables are related to key locations in the solution: the main root directory, the content directory, the configuration directory, the logging directory, and so on. These variables will be used in the solution to refer to other artifacts.

Note that many of these global properties inherit from the `ROOT_DIR` or `CONTENT_DIR` variables. The reason for still having them at `kettle.properties` level is that this gives you the flexibility to define a different path (that is, not a subfolder from `CONTENT_DIR`) and also makes the solution clearer by having dedicated variables for dedicated functionality.

Sharing Database Connections with shared.xml

Since you will not be using the Pentaho Repository in your local environment, you need to have an alternative available for sharing database connections between developers, jobs, and transformations. When you work file-based, you can use `shared.xml`.

The `shared.xml` file holds the configuration details of the database connections shared among all the developers and all PDI jobs and transformations per environment. All development will be done using the Git repository, so you can use the `shared.xml` file to share database connections when working file-based with PDI. Database connections are set up environment-specific.

This file is only useful within your local environment when using Spoon. Once the ETL code is deployed to the Pentaho Server, the database connections will be deployed to the Pentaho Repository.

Since this file is part of the common `.kettle` directory, all developers have access to the same database connections. Developers should only use shared database connections in their jobs and transformations.

Spoon GUI Options in .spoonrc

The `.spoonrc` file holds the Spoon GUI options which you can set through **Tools** > **Options** in Spoon. We recommend using a canvas grid size of 32 pixels and, if desired, using a specific tab color indicating the project environment (such as `local=green`, `prod=red`).

The canvas grid size of 32 pixels guarantees that no developer can change the layout of a job or transformation by simply opening it. The following properties set this behavior:

```
ShowCanvasGrid=Y
CanvasGridSize=32
```

Since all developers are starting Spoon through the environment specific `spoon.bat/sh` script which makes sure the right environment details get loaded, you can avoid mistakes by making it visually clear which Spoon you are using. The following properties make your tab green:

```
TabColorR54=0
TabColorG54=255
TabColorB54=64
```

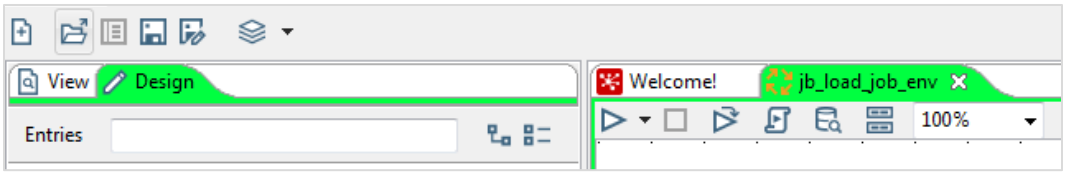

Figure 1: Spoon Tab Color

Specific Variables in the properties Folder

Where `kettle.properties` holds the global variables for the project, the `properties` files within the `properties` folder hold the project variables that are more subject to change and the variables dedicated to a specific job or transformation within the project.

The `kettle.properties` file provides an easy way to introduce global variables to PDI, but the downside is that any changes to the file require a Pentaho Server restart for the changes to be picked up by the server. That is why you would only use it for global variables that have a small chance of changing over the lifetime of the projects.

The `properties` files will be loaded at job or transformation execution time, allowing you to introduce new variables in a dynamic way. The `properties` files also allow you to introduce more structure by having separate `properties` files per project or job.

At a minimum, we recommend that you introduce project and job `properties` files. Look at it this way: when the `sales_dwh_staging` job of your `sales_dwh` project gets executed, the `kettle.properties` global variables that are available by default will make sure the global variables get introduced. This way, the main directories are set, and logging settings are defined globally. We will also load the project `properties` file (`sales_dwh.properties`) and the job properties file (`sales_dwh_staging.properties`). This allows you to introduce variables valid at the project level

(shared among jobs within the project) and variables valid at the specific job level. You can define all the levels you need and create `properties` files accordingly.

Project Specific Variables in project.properties

In addition to other variables, all `project.properties` files introduce `_HOME` variants of the `_DIR` variables in `kettle.properties`, making the variables project specific. For example, the `sales_dwh.properties` file introduces the following mandatory variables:

Table 10: Mandatory Variables in `sales_dwh.properties`

Variable	Description	Example
PROJECT_NAME	The name of the project within the ETL repository	`sales_dwh`
CONTENT_HOME	The folder that houses the ETL artifacts for this specific project. Since `${CONTENT_DIR}` indicates whether to use a filesystem or Pentaho Repository, this variable can simply inherit from that variable.	• local: `${CONTENT_DIR}/sales_dwh/content-pdi` • server: `${CONTENT_DIR}/sales_dwh`
LOG_HOME	The file logging location of the project, inheriting from `${LOG_DIR}`.	`${LOG_DIR}/sales_dwh/log`
FILE_MGMT_HOME	The `file_mgmt` location of the project, inheriting from `${FILE_MGMT_DIR}`	`${FILE_MGMT_DIR}/sales_dwh/file-mgmt`
SQL_HOME	The SQL location of the project, inheriting from `${ROOT_DIR}`	`${ROOT_DIR}/sales_dwh/sql`
SCRIPT_HOME	The script location of the project, inheriting from `${ROOT_DIR}`	`${ROOT_DIR}/sales_dwh/scripts`

Apart from these mandatory variables, the `project.properties` files can introduce any variable that is needed at the project level.

*Note that although you defined `${PROJECT_NAME}`, you still need to hardcode the project name in all other variables. That is because the **Set Variables** job entry used later on in the DI framework declares those variables in parallel and not sequentially.*

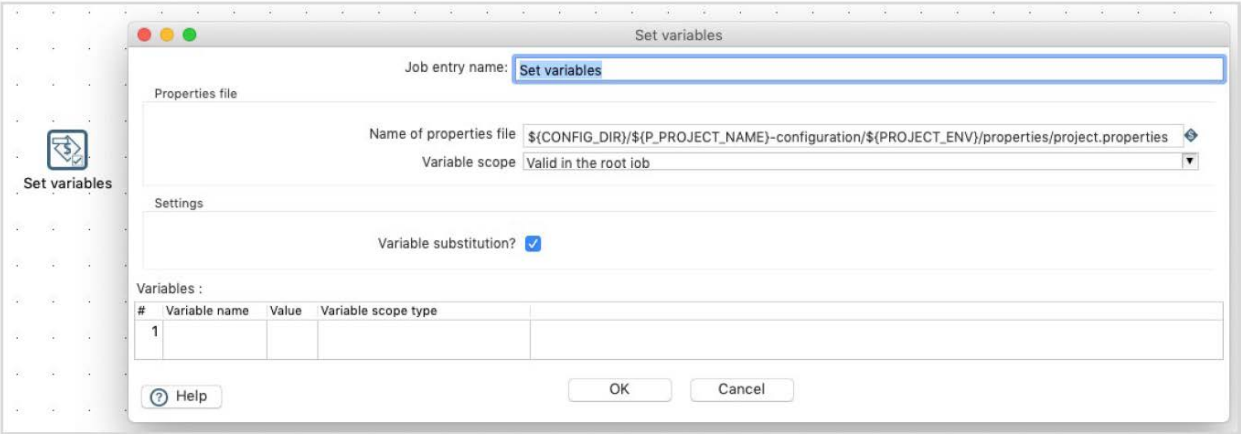

Figure 2: Set Variables Job Entry

Job Specific Variables in job.properties

Your projects will most likely contain multiple main jobs. While your `project.properties` can introduce variables valid for the whole project, you can use the `job.properties` files to introduce variables for the specific job. There are no mandatory variables at this level. The `job.properties` files will have the same name as the name of their job.

PDI Start Scripts

Before you start Spoon, Kitchen, Pan, or Pentaho Server for a specific project environment, you need to run environment-specific start scripts that will set some environment variables, change the `KETTLE_HOME` location, change the `metastore` location, and load the right environment properties before starting the actual PDI clients or server. It is important that we start the client tools or the Pentaho Server this way to have them configured for the right environment. Four scripts are available per environment in the `config` directory:

- `spoon.bat/sh`
- `kitchen.bat/sh`
- `pan.bat/sh`
- `start-pentaho.bat/sh` (only available for the server environments)

These scripts are wrappers for the actual Spoon and Pentaho Server scripts, and are part of the Pentaho installation directory.

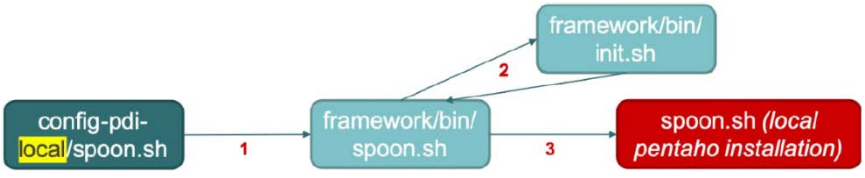

Figure 3: Scripts Hierarchy

Developers should use the `spoon.bat/sh` script, located in the `config-pdi-local` directory, to start Spoon on their local development machines. Starting Spoon this way gives it the right configurations to work with, pointing to the `.kettle` folder from the `config-pdi-local` environment, and all other specific settings for the local environment.

See the **`sales_dwh-configuration/config-pdi-local/spoon.sh` script**:

```
#!  /bin/sh
#PROJECT_NAME = current project name, eg. sales_dwh
#PROJECT_ENV = current folder name, eg. config-pdi-local

#Determine the PROJECT_NAME and _ENV configuration based on parent
directory
export PROJECT_NAME=$(basename $(dirname $PWD) -configuration)
#removing -configuration from dir name
export PROJECT_ENV=$(basename $PWD)

echo "***" Setting PROJECT_NAME to $PROJECT_NAME "***"
echo "***" Setting PROJECT_ENV to $PROJECT_ENV "***"

#BASE_CONFIG_DIR is used later in init.sh to call back to this config-dir
export BASE_CONFIG_DIR=$PWD

#Capture current dir to switch back after execution
export ORIG_DIR=$PWD

cd "$PWD"/../../framework/bin
#Call framework common script
sh spoon.sh
cd $ORIG_DIR
```

Alternatively, see the **`sales_dwh-configuration/config-pdi-local/spoon.bat` script**:

```
@echo off

REM Determine the PROJECT_NAME and _ENV configuration based on parent
directory
for %%a in ("%~dp0\.") do set "PROJECT_NAME_TEMP=%%~nxa"
set PROJECT_NAME=%PROJECT_NAME_TEMP:-configuration=%

for %%a in ("%~dp0\.") do set "PROJECT_ENV=%%~nxa"

ECHO *** Setting PROJECT_NAME to "%PROJECT_NAME%" ***
ECHO *** Setting PROJECT_ENV to "%PROJECT_ENV%" ***

REM BASE_CONFIG_DIR is used later in init.bat to call back to this config-
dir
set BASE_CONFIG_DIR=%CD%

REM Capture current dir to switch back after execution
set ORIG_DIR=%CD%
CD %~dp0\\..\\..\\framework\\bin
REM Call framework common script
CALL spoon.bat %*
cd %ORIG_DIR%
```

The scripts simply set two environment variables and call the spoon script located within the `framework` Git repository (`bin` directory that we will discuss in detail in the Data Integration Framework section.

See the **framework/bin/spoon.sh script**:

```
. ./init.sh

sh $KETTLE_CLIENT_DIR/spoon.sh $OPT "$@"
```

Alternatively, see the **framework/bin/spoon.bat script**:

```
CALL init.bat

CALL %KETTLE_CLIENT_DIR%\Spoon.bat %OPT% %*
```

The scripts call the `init` script located in the `bin` directory and start the actual `spoon` script located within the local Pentaho installation directory.

See the **framework/bin/init.sh script**:

```
#!/bin/bash

# Set Environment
ROOT_DIR="$PWD"/../..
PROJECT_ENV="${PROJECT_ENV:-config-pdi-local}"

echo "***" Setting ROOT_DIR to "$ROOT_DIR" "***"
echo "***" Running with "$PROJECT_ENV" environment settings "***"

# Load Environment Configuration
#. $ROOT_DIR/$PROJECT_NAME-configuration/$PROJECT_ENV/env.conf
. $BASE_CONFIG_DIR/env.linux.conf
echo "***" Setting PENTAHO_HOME to $PENTAHO_HOME "***"

# Set Additional Variables
export KETTLE_CLIENT_DIR="${KETTLE_CLIENT_DIR:-$PENTAHO_HOME/design-
tools/data-integration}"
export KETTLE_HOME=$BASE_CONFIG_DIR
export KETTLE_META_HOME=$BASE_CONFIG_DIR
export OPT="$OPT -DPENTAHO_METASTORE_FOLDER=$KETTLE_META_HOME –
DROOT_DIR=$ROOT_DIR -DPROJECT_ENV=$PROJECT_ENV –
DPROJECT_NAME=$PROJECT_NAME"

echo "***" Setting KETTLE_CLIENT_DIR to $KETTLE_CLIENT_DIR "***"
echo "***" Setting KETTLE_HOME to $KETTLE_HOME "***"
echo "***" Setting KETTLE_META_HOME to $KETTLE_META_HOME "***"
```

Alternatively, see the **framework/bin/init.bat script**:

```
@echo off

REM Set Environment
REM ~dp0 has \ at end already
SET ROOT_DIR=%~dp0..\..

IF "%PROJECT_ENV%"=="" (SET PROJECT_ENV=config-pdi-local)

ECHO *** Setting ROOT_DIR to "%ROOT_DIR%" ***
ECHO *** Running with "%PROJECT_ENV%" environment settings ***
```

```
REM Load Environment Configuration
FOR /F "usebackq delims=" %%x IN ("%BASE_CONFIG_DIR%\env.windows.conf") DO
SET %%x
ECHO *** Setting PENTAHO_HOME to %PENTAHO_HOME% ***

REM Set Additional Variables
IF "%KETTLE_CLIENT_DIR%"=="" (SET
KETTLE_CLIENT?DIR=%USERPROFILE%\%PENTAHO_HOME%\data-integration)
SET KETTLE_HOME=%BASE_CONFIG_DIR%
SET KETTLE_META_HOME=%BASE_CONFIG_DIR%
SET OPT="-DPENTAHO_METASTORE_FOLDER=$KETTLE_META_HOME%" "-
DROOT_DIR=%ROOT_DIR%" "-DPROJECT_ENV=%PROJECT_ENV%" "-
DPROJECT_NAME=%PROJECT_NAME%"

ECHO *** Setting KETTLE_CLIENT_DIR to %KETTLE_CLIENT_DIR% ***
ECHO *** Setting KETTLE_HOME to %KETTLE_HOME% ***
ECHO *** Setting KETTLE_META_HOME to %KETTLE_META_HOME% ***
```

The scripts set the `ROOT_DIR` variable, load the `env.conf` file variables, set the location of the Kettle clients, set the correct `KETTLE_HOME` and location of the `metastore`, and set the right JVM settings.

The other client and server scripts work in a similar fashion.

Use the `start-pentaho.bat/sh` script to start the Pentaho Server on the environments that use the Pentaho Server for job execution. Starting the Pentaho Server this way gives it the right environment configuration. The actual script is like the one for Spoon.

Password Encryption

Storing passwords in plaintext inside properties files poses risks. There are two ways to secure passwords in PDI:

- Kettle obfuscation
- Advanced Encryption Standard (AES) encryption

Kettle obfuscation is applied by default, while AES encryption requires additional configurations.

For greater security, we recommend using the AES standard. More information on how to configure PDI for AES is available in AES Security.

The password security method you choose is applied to all passwords, including those in database connections, transformation steps, and job entries.

For demonstration purposes only, the remainder of this document and all included samples will have passwords in plaintext and will not have password encryption enabled.

Security

The security of the DI solution is composed of multiple layers:

- **AES Password Encryption**: As discussed in Configuration Management, no plaintext passwords will be stored in configuration files.

- **Git repository and branch access is password restricted**. Developers only have access to the projects folders or the configuration folders of the environments that they will be developing and testing on.
- **Pentaho Repository access is password restricted**. Developers only have access to those Pentaho Repositories on the environments that they will be developing and testing on.
- **Machine access is password restricted**. Developers only have access to those environments that they will be developing and testing on.

Configuration Repository Structure

In summary, you now have the following structure for your `sales_dwh` project's Git configuration repository:

```
|-- sales_dwh-configuration
|      |-- config-pdi-local
|      |      |-- .kettle
|      |      |      |-- kettle.properties
|      |      |      |-- shared.xml
|      |      |      |-- repositories.xml
|      |      |      |-- .spoonrc
|      |      |      |-- .languageChoice
|      |      |-- properties
|      |      |      |-- project.properties
|      |      |      |-- jb_main_sales_dwh_staging.properties
|      |      |-- metastore
|      |      |-- env.linux.conf
|      |      |-- env.windows.conf
|      |      |-- spoon.bat/sh
|      |      |-- kitchen.bat/sh
|      |      |-- pan.bat/sh
|      |-- config-pdi-dev
|      |      |-- .kettle
|      |      |      |-- kettle.properties
|      |      |      |-- shared.xml
|      |      |      |-- repositories.xml
|      |      |      |-- .spoonrc
|      |      |      |-- .languageChoice
|      |      |-- properties
|      |      |      |-- project.properties
|      |      |      |-- jb_main_sales_dwh_staging.properties
|      |      |-- metastore
|      |      |-- env.linux.conf
|      |      |-- env.windows.conf
|      |      |-- spoon.bat/sh
|      |      |-- kitchen.bat/sh
|      |      |-- pan.bat/sh
|      |      |-- start-pentaho.bat/sh
|      |-- config-pdi-test
…
|      |-- config-pdi-prod
…
```

Data Integration Framework

Now that you have seen the basic concepts of content and configuration management, take a first look at the DI framework solution that builds on those best practices and that illustrates how to implement all other project setup and governance best practices that will be covered later in this document.

The DI framework is a set of jobs and transformations that act as a wrapper around your actual DI solution, taking care of all setup, governance, and control aspects of your solution. By abstracting these concepts from the actual DI solution, the development team can focus on the actual jobs and transformations.

You can find details on these topics in the following sections:

Dedicated Framework Repository

So far, we have been talking about two main Git repositories per project: `etl` and `configuration`.

The DI framework, which is a set of jobs and transformations, is stored in its own Git repository, `framework`. This repository has the same structure as any other project under the `etl` repository:

```
|-- framework
|      |-- content-pdi
|      |-- bin
|      |-- sql
|      |-- documentation
```

The benefits of having the framework stored in a separate Git repository include:

- You can work on it by itself without affecting the actual DI solution.
- You can apply a different release cycle to it.
- You can apply different security roles to it.

In the PDI Start Scripts section, we discussed the `bin` folder in the `framework` repository. The scripts in this folder abstract the core logic from the scripts in the `config` directories. By abstracting it here, the logic can be maintained in a single location and is not repeated across all `config` scripts.

In addition to having a dedicated repository, you also have two `kettle.properties` variables indicating the framework's location:

```
FRAMEWORK_DIR=${ROOT_DIR}/framework
FRAMEWORK_HOME=${ROOT_DIR}/framework/content-pdi
```

This is based on the same reasoning as before: the `_DIR` variable is available to reference all artifacts that will stay on the filesystem (like scripts), while the `_HOME` variable can reference the DI content that will be deployed to the Pentaho Repository.

You also have a dedicated `properties` file, called `framework.properties`.

For now, the only important variables here are the ones defining the `pdi_control` database connection. The `pdi_control` connection contains the `job_control` table. This table holds many critical roles within the framework and always reflects the current status per job in the actual DI solution.

Table 11: Control Variables

Category	Variables	Description

`pdi_control` database connection variables	`PDI_CONTROL.HOST`
	`PDI_CONTROL.PORT`
	`PDI_CONTROL.DB`
	`PDI_CONTROL.SCHEMA`
	`PDI_CONTROL.USER`
	`PDI_CONTROL.PASS`
		These variables will be used in the `shared.xml` database connection definition of the `pdi_control` database connection.

`job_control` table	`PDI_CONTROL.TABLE`	The name of the `job_control` table used

Concepts

Before we dive into the details of the DI framework, we need to discuss some terms:

Main Job and Work Units

Your projects will most likely consist of several **main jobs**, at which level a schedule will be defined.

Work units are the jobs or transformations that are being executed in sequence by the main job. They act as a checkpoint at which job restartability gets implemented. Think of them as multiple stages in your main job that you must go through. The work unit succeeds or fails in its entirety.

Job Restartability

The framework offers **job restartability**, not data restartability. However, we do make variables available whose values are populated according to whether the previous execution was successful. This way, the framework lets you know when to address data restartability within the work unit. Everything within the work unit must be restartable.

Job restartability at a high level works in the following way:

- If the previous main job executed successfully, all work units will execute normally.
- If the previous job failed, you will skip the work units that executed successfully, rerun the work unit that failed, and have a normal execution for those work units that did not yet get executed in the previous run.

Tracking Current Status with the job_control Table

The framework uses a control table called `job_control` to track the current status of all jobs and work units running in the solution. In addition to the status, this table also tracks additional metadata like start and end times, the IP address and hostname where the job is running, and the process ID on the operating system. You will also use this table for implementing your restartability solution that will be discussed later.

Since this table keeps track of the current status, you know which work units of the main job were executed successfully and which were not. When jobs need to restart because of an error in their execution, they can skip those work units that already executed successfully, and rerun those that had an error in their execution:

Table 12: `job_control` Table Status Information

Data	Description
`batch_id`	Unique ID of the job, providing a link to the PDI logging tables
`jobname`	Name of the job
`work_unit`	Name of the work unit
`status`	Status of the job or work unit
`project`	Project where the job is part of
`starttime`	Start time of the job or work unit
`logtime`	Last update time of record
`ip_address`	IP address of server where job is running
`hostname`	Host name of server where job is running
`pid`	ID of Java process executing the PDI JVM

Let's take a closer look at some of the jobs and transformations within the framework repository. The `content_pdi` folder has the following subfolders:

Table 13: List of `content_pdi` Subfolders

Folder	Description
`control`	All jobs and transformations related to updating the `job_control` table
`execution`	All jobs and transformations related to the execution of main jobs and work units
`utilities`	All jobs and transformations related to minor functionalities within the framework
`developer-tools`	All jobs and transformations that support the developer in using the framework

Triggering Framework Job Executions with jb_launcher Job

One of the main jobs in your `sales_dwh` project is the `jb_main_sales_dwh_staging` job, which is responsible for loading all `CSV` extracts into their dedicated staging tables. Each individual staging transformation (customer and product) will be treated as a work unit.

Since your project uses variables defined in `properties` files in the `configuration/config-pdi-local/properties` folder, you cannot simply run the main job since these variables values are not yet known. Therefore, all main job executions within the framework are triggered through the `jb_launcher` job (`framework/content-pdi/execution/jb_launcher.kjb`):

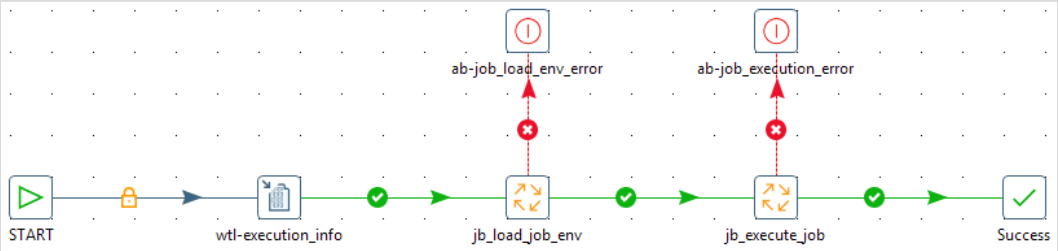

Figure 4: `jb_launcher.kjb`

The execution starts with the `jb_launcher` framework job, which has input parameters to define which main job to execute: `P_PROJECT_NAME` and `P_JOB_NAME`.

`jb_load_job_env` (`${FRAMEWORK_HOME}/utilities/jb_load_job_env.kjb`) is executed to load the job environment:

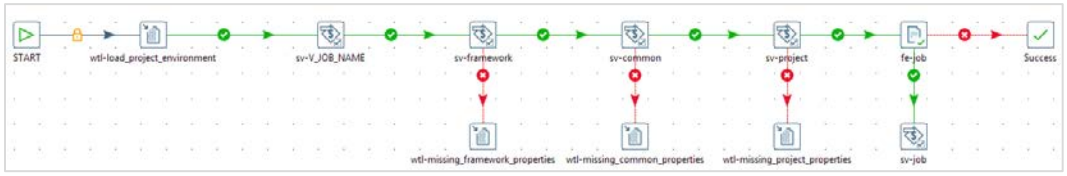

Figure 5: `jb_load_job_env.kjb`

This job creates `V_JOB_NAME`, and loads the following:

- `framework.properties` (`${CONFIG_DIR}/properties/framework.properties`)
- `${P_PROJECT_NAME}.properties` (`${CONFIG_DIR}/properties/${P_PROJECT_NAME}.properties`)
- `${V_JOB_NAME}.properties` (if this file exists) (`${CONFIG_DIR}/properties/${V_JOB_NAME}.properties`)

All these variables are loaded with their variable scope set to **Valid in the root job**.

`jb_execute_job` executes the actual `jb_main_sales_dwh_staging` main job once all variables are loaded, calling the main job with `${CONTENT_HOME}/${P_JOB_NAME}.kjb`. `${CONTENT_HOME}` points to the current project's `content-pdi` folder. This variable is set by the `project.properties` file.

Main Job Template

All main jobs within the project (similar to `jb_main_sales_dwh_staging`) will follow the same template, using elements of the framework:

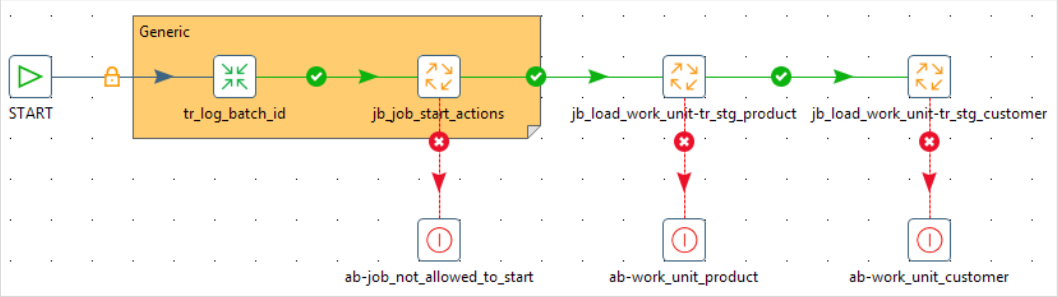

Figure 6: `jb_main_sales_dwh.kjb`

1. Execute `tr_log_batch_id` (`${FRAMEWORK_HOME}/control/tr_log_batch_id.ktr`). This transformation guarantees that the following variables get created:

Table 14: Variables Created in `tr_log_batch_id.ktr`

Variable	Description
`V_BATCH_ID`	Main job `batch_id` (coming from job log table)
`V_IP_ADDRESS`	The IP address of the host where the job gets executed
`V_HOSTNAME`	The hostname of the host where the job gets executed
`V_PID`	The process ID of the job that gets executed

2. Next, execute `jb_job_start_actions`, located at (`${FRAMEWORK_HOME}/control/jb_job_start_actions.kjb`), to verify that the main job can start.

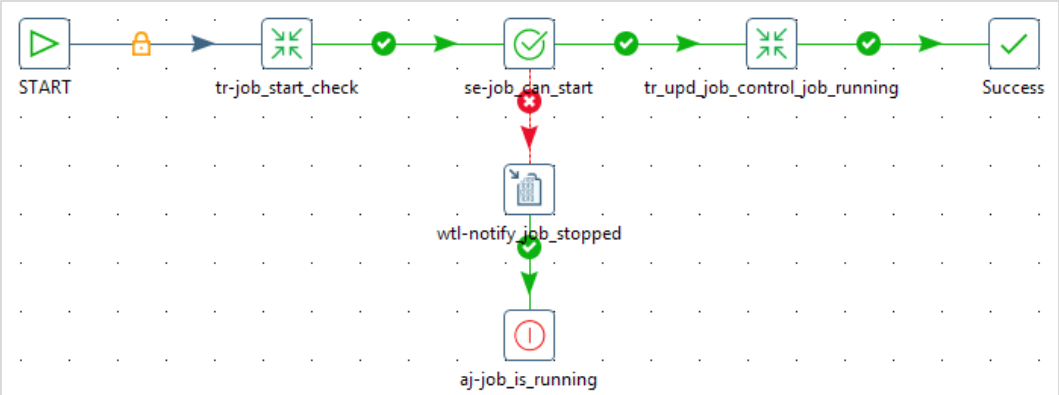

Figure 7: `jb_job_start_actions.kjb`

If the job can start, the main job status is updated with `tr_upd_job_control_job` (`${FRAMEWORK_HOME}/control/tr_upd_job_control_job.ktr`). If it cannot start, the execution of the main job is aborted.

When the main job is allowed to start, it executes the work units in sequence through the `jb_work_unit` wrapper job (`${FRAMEWORK_HOME}/execution/jb_work_unit.kjb`).

Development Variable Workaround with jb_set_dev_env Job

Since the jobs and transformations of the actual DI solution make use of variables defined through project and job `properties` files, this poses a problem during development.

These variables are only declared once the main job gets executed through the `jb_launcher` job, and those properties files are loaded before the actual job gets executed. That means that during development, those variables will not be available.

This is not the case with variables defined by `kettle.properties`, since these are available by default in the PDI JVM. Therefore, you need a workaround during development.

This workaround is available as the `jb_set_dev_env` job (`framework/content-pdi/developer-tools/jb_set_dev_env.kjb`).

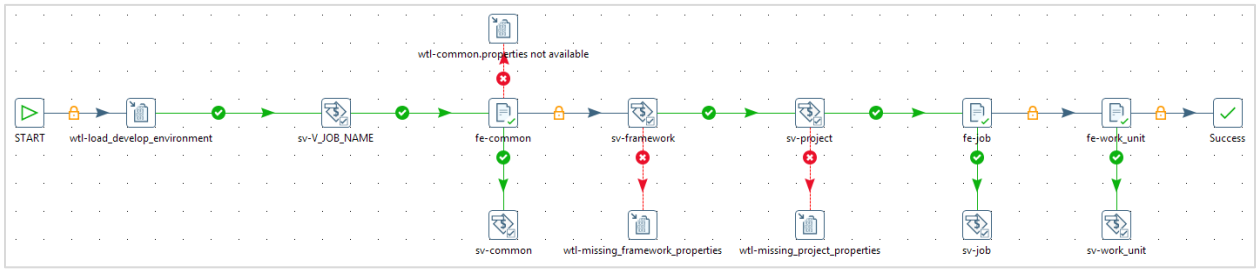

Figure 8: `jb_set_dev_env.kjb`

This job requires three input parameters:

Table 15: Job Input Parameters

Parameter	Description
P_PROJECT	Name of the project
P_JOB_NAME	Name of the job (no extension needed)
P_WORK_UNIT_NAME	Name of the work unit (no extension needed)

The variable **scope** used to set these variables needs to be **Valid in the Java Virtual Machine**, or the variables will not be available for all jobs and transformations in Spoon. This example where we set development variables will most likely be the only place we will use this variable scope (valid in the JVM).

Run this job as soon as you open Spoon, without having any other jobs or transformations open, because only jobs or transformations that you open after *running this job will pick up the newly defined variables.*

Framework Repository Structure

So far, we have introduced the basic functionality of the DI framework, and the next sections go into more detail around the additional functionality of the framework. This overview shows the jobs and transformations in the framework:

```
|-- framework
|      |-- content-pdi
|      |      |-- execution
|      |      |      |-- jb_launcher.kjb
|      |      |      |-- jb_work_unit.kjb
|      |      |-- control
|      |      |      |-- jb_job_start_actions.kjb
|      |      |      |-- tr_job_start_check.ktr
|      |      |      |-- tr_log_batch_id.ktr
|      |      |      |-- tr_upd_job_control_job.ktr
|      |      |      |-- tr_upd_job_control_work_unit.ktr
|      |      |      |-- tr_work_unit_start_check.ktr
|      |      |-- utilities
|      |      |      |-- jb_load_job_env.kjb
|      |      |      |-- jb_load_work_unit_env.kjb
|      |      |      |-- jb_mail.kjb
|      |      |-- developer-tools
|      |      |      |-- jb_set_dev_env.kjb
|      |-- bin
|      |-- sql
|      |-- documentation
```

Logging and Monitoring

Every process executed within PDI has feedback information related to workflow logging. This gives details about what is happening during execution. Logging can:

- Provide relevant information whenever a process execution has an error, such as which steps are failing, and trace with the main error description.
- Give information about a workflow if it has decision splits.
- Detect bottlenecks and substandard performance steps based on a procedure's duration; for example, stored execution times can be used to detect if a process is taking longer than usual.
- Show the status of currently running processes. Logs provide information about when the process started, where it is currently, and data related to its status.
- Provide traceability of what has been done, and when.

More information on logging can be found at Logging, Monitoring, and Performance Tuning for Pentaho.

Pentaho Data Integration has two main options for logging: log entries (file logging) and database logging. You can find details on these and other topics in the following sections:

- File Logging
- Database Logging
- Exception Handling
- Launching DI Solution Work Units with jb_work_unit Job

File Logging

For debugging purposes, we recommend that every main job's execution redirect all logging related to the execution of this job to a single log file per execution and per job.

In the DI framework, the log messages generated when a job is being executed are written to a file located inside the `log` folder of the project. The `project.properties` `LOG_HOME` variable determines the `log` folder's location, while the name of the job and the time it is executed determine the name of the log file.

This logging behavior is configured in the `jb_launcher` job at the `jb_execute_job` job entry's logging tab:

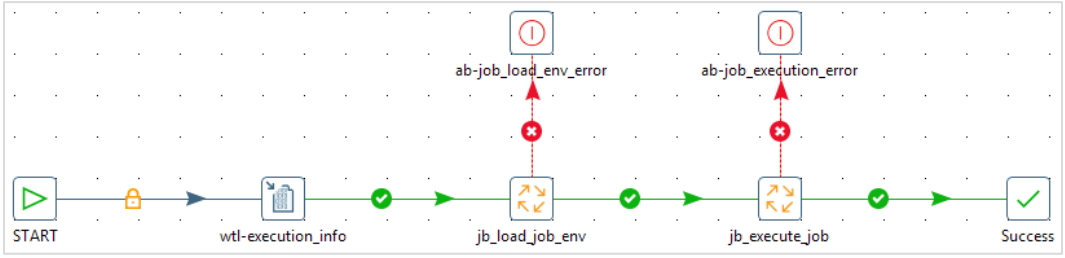

Figure 9: `jb_launcher.kjb`

Here is a closer look at the `jb_execute_job` job entry settings:

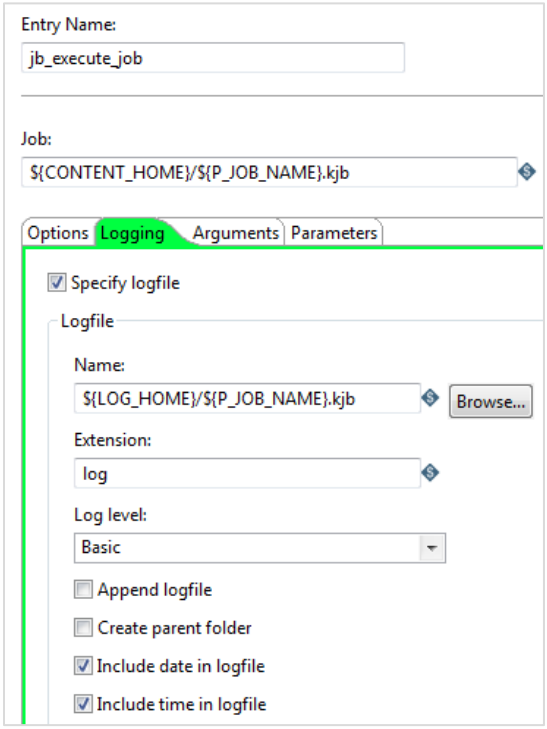

Figure 10: `jb_execute_job` Job Entry Logging Tab

Logging Levels

PDI lets you establish different levels of file logging verbosity depending on your needs:

Table 16: File Logging Verbosity

Logging Level	Description
Nothing	Logging is enabled but does not record any output
Error	Only shows error lines
Minimal	Only uses minimal logging, informing about workflow status
Basic	**Default** setting: Shows information related to every step
Detailed	For troubleshooting, gives detailed login output
Debug	Detailed output for debugging, not for use in production
Row Level	Logging at row level detail, generating a huge among of log output

In general, logging levels should be lower in a production or quality assurance (QA) environment, but can be higher in a development or non-production environment.

Specify these levels when developing and running by using Spoon (PDI client), or by redirecting the main job logging feedback to a dedicated log file for this solution.

For this framework's file logging, the level used will always be set to **Basic**.

For Spoon, the logging output shows in the Spoon Logging tab. When running Spoon, you can change the standard log level (**Basic**) with the **Run Options**:

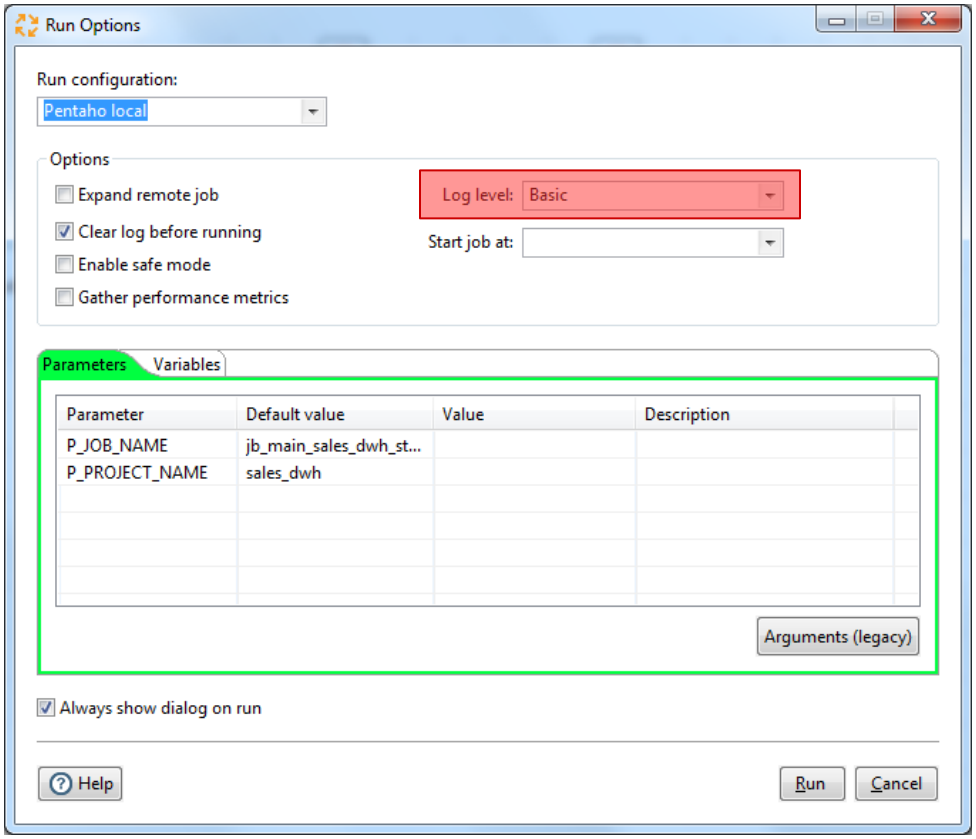

Figure 11: Run Options

Redirect Output to Kettle Logging

Change the following `kettle.properties` variables to `Y` to redirect all output to PDI logging destinations:

Table 17: `kettle.properties` Variables for Redirecting Output

Variable	Value
KETTLE_REDIRECT_STDERR	Y
KETTLE_REDIRECT_STDOUT	Y

These variables are set to `N` by default. Turning them to `Y` gives `STDERR` and `STDOUT` more useful information for logging and debugging errors.

Central Logging Directory

By default, the framework's file logging is configured to be project specific. For example, all main jobs of the `sales_dwh` project will be writing their logging output to the `sales_dwh/log` folder. If you prefer to have your file-logging configured in a central manner, where all main jobs are logging to the same top-level folder, you have the option to do so via configuring the framework's logging variables, `LOG_DIR` (`kettle.properties`) and `LOG_HOME` (`project.properties`):

Default project-specific logging configuration:

- `LOG_DIR=${ROOT_DIR}`
- `LOG_HOME=${LOG_DIR}/sales_dwh/log`

Central logging configuration example:

- `LOG_DIR=${ROOT_DIR}/log`
- `LOG_HOME=${LOG_DIR}/sales_dwh`

Database Logging

PDI is also capable of writing structured logging feedback to log tables. The framework redirects structured database (DB) logging information to a central `pdi_logging` database.

The DB logging configurations are applied globally at the `kettle.properties` level. This allows you to specify the logging tables that each environment uses.

The `kettle.properties` variables used by the framework are the following:

Table 18: `kettle.properties` Variables Used by Framework

Variable Type	Variable	Description
`pdi_logging` Database Connection Variables	**`PDI_LOGGING.HOST`** **`PDI_LOGGING.PORT`** **`PDI_LOGGING.DB`** **`PDI_LOGGING.SCHEMA`** **`PDI_LOGGING.USER`** **`PDI_LOGGING.PASS`**	These variables are used in the `shared.xml` database connection definition of the `pdi_logging` connection
Job DB Logging Variables	**`KETTLE_JOB_LOG_DB`**	The Job log default database connection for all jobs
	`KETTLE_JOB_LOG_SCHEMA`	The Job log default schema for all jobs
	`KETTLE_JOB_LOG_TABLE`	The Job log default table for all jobs
Transformation DB Logging Variables	**`KETTLE_TRANS_LOG_DB`**	The transformation log default database connection for all transformations
	`KETTLE_TRANS_LOG_SCHEMA`	The transformation log default schema for all transformations
	`KETTLE_TRANS_LOG_TABLE DB`	The transformation log default table for all transformations

Variable Type	Variable	Description
Channel DB Logging Variables *Provide hierarchical information between job and transformation logging*	**KETTLE_CHANNEL_LOG_DB**	The channel log default database connection for all transformations and jobs
	KETTLE_CHANNEL_LOG_SCHEMA	The log default schema for all transformations and jobs
	KETTLE_CHANNEL_LOG_TABLE	The log channel default table for all transformations and jobs
	KETTLE_TRANS_LOG_SCHEMA	The transformation log default schema for all transformations

By specifying this at a global `kettle.properties` level, you do not need to configure those details anymore at each job or transformation with its logging transformation or job properties. You can leave those settings blank.

We also recommend you use the following additional database logging `kettle.properties` configuration:

Table 19: Database Logging `kettle.properties` Configuration

Variable	Value
KETTLE_LOG_SIZE_LIMIT	100

This makes sure only the last 100 lines of the console logging get written to the `LOG_FIELD` of the job and transformation database logging tables. This should be fine for the database logging, since all logging output is available for debugging in the main job file logging.

Job Database Logging

One of the advantages of using job database logging is that this gives you a unique batch ID (`ID_JOB` column) that is increased by one for each run of a job.

In the framework, you reuse this unique batch ID in both the `job_control` table (`batch_id` column) and the job file logging. Having this unique batch ID in all those places gives you a way of connecting them, making debugging easier. You could also reuse this ID to attach them to records in your tables to associate the record with the job that inserted or updated the record.

Table 20: Unique Batch ID

Field name	Description
ID_JOB	The batch ID. It's a unique number, increased by one for each run of a job

In the main job's job properties, make sure to check the **Pass batch ID?** checkbox to make this `batch_id` available to your jobs and transformations. This is the only place where you need to do this.

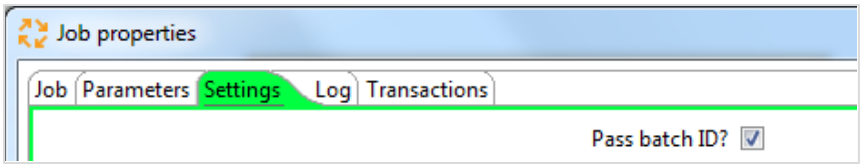

Figure 12: Main Job Properties

You can store this batch ID in a variable so that it can be reused easily within the framework. Do this with the `tr_log_batch_id` transformation, which is the first transformation that gets executed within the main job (see `jb_main_sales_dwh.kjb`), located at (`framework/content-pdi/control/tr_log_batch_id.ktr`).

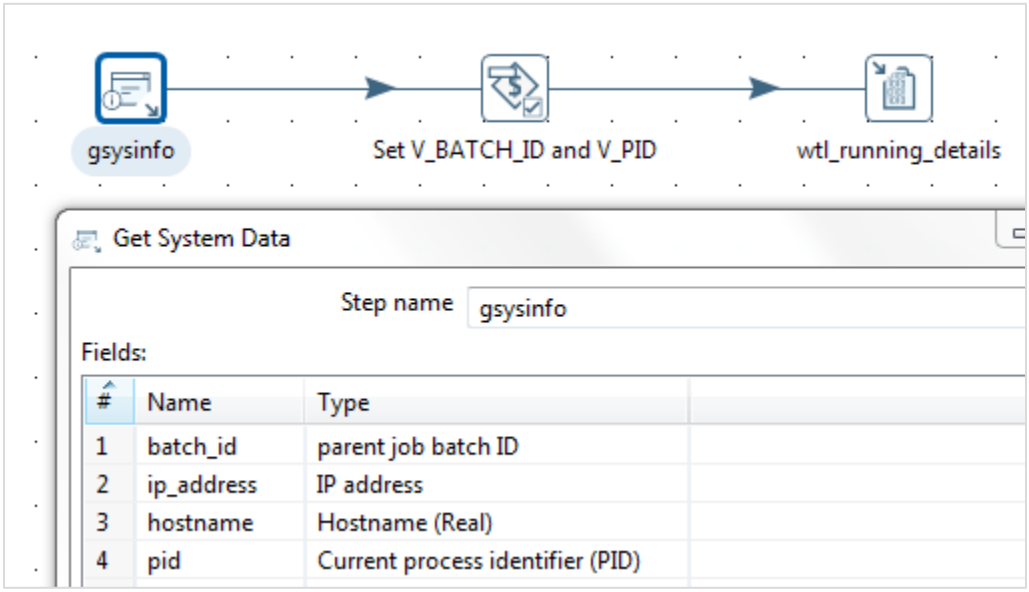

Figure 13: `tr_log_batch_id.ktr`

Transformation Database Logging

For both the job and channel database logging, you can keep the default configurations at the **logging tab** of the job or transformation properties.

For the transformation database logging, however, you have the option to configure additional logging detail by specifying a **step name** for the following logging fields. Transformation logging is at the level of the transformation and the `LINES_` metrics are at the level of a step; therefore, a step name is required:

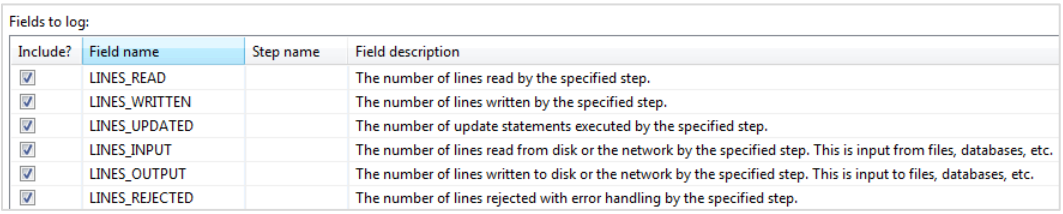

Figure 14: Transformation Database Logging

Table 21: Step Names for Logging Fields

Field Name	Description	Step Name
LINES_READ	The number of lines read by the specified step	Only used when applicable
LINES_WRITTEN	The number of lines written by the specified step	Only used when applicable
LINES_UPDATED	The number of update statements executed by the specified step	The main output step should be used here when applicable
LINES_INPUT	The number of lines read from disk or the network by the specified step This is input from files, databases, and so on.	The main input step should be used here
LINES_OUTPUT	The number of lines written to disk or the network by the specified step. This is input to files, databases, and so on.	The main output step should be used here
LINES_REJECTED	The number of lines rejected with error handling by the specified step	Only used when applicable

For example, by specifying the step name for those metrics, you can capture how many records you read with your **Text file input** step and how many records you wrote to your staging table with the **Table output** step during the execution of your transformation.

Configuring these step names makes that additional information available in your transformation log table.

Logging Concurrency

For database logging to work correctly, PDI needs to generate a unique integer ID at the very start of the transformation or job. This batch ID will then be used in various places throughout the life of the transformation or job.

However simple this may seem, it can be surprisingly tricky to use a database to generate a unique ID. The problem is mostly caused by inadequate or incorrect locking by the various databases. To make matters worse, every database behaves differently when it comes to locking tables. Configuring Log Tables for Concurrent Access describes two options on configuring the logging tables for concurrent access.

If you don't set this up, there's a slight chance you can run into table deadlocks if two jobs or transformations execute at the same moment and lock the same logging table. Over enough time, it will happen, and it will cause the job or transformation to hang until you manually kill it.

Exception Handling

Proper error and exception handling should be part of your DI solution design from the early stages. What happens when something in the DI process fails? The exact answer depends on the source of failure and whether you are in control or not.

Error handling in the framework and the actual DI solution can take many shapes and forms:

- Concurrency Checks
- Dependency Management
- Job Restartability
- Data Restartability
- Transformation and Job Error Handling

Concurrency Checks

If the main job is still running when the next execution is scheduled, you do not want to launch the job again. This conflict can occur due to abnormal circumstances, such as having more data to process than normally. The framework has concurrency checks in place.

A concurrency check is performed with the `jb_job_start_actions` job that is (`${FRAMEWORK_HOME}/control/jb_job_start_actions.kjb`) executed as part of the main job.

For example, here is the main job:

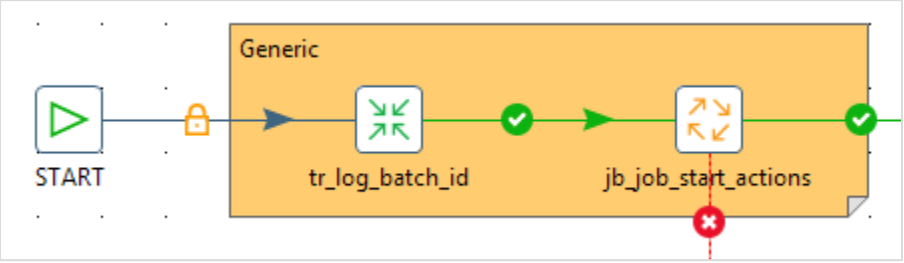

Figure 15: Main Job

Next, here is the `jb_job_start_actions` job from within the main job:

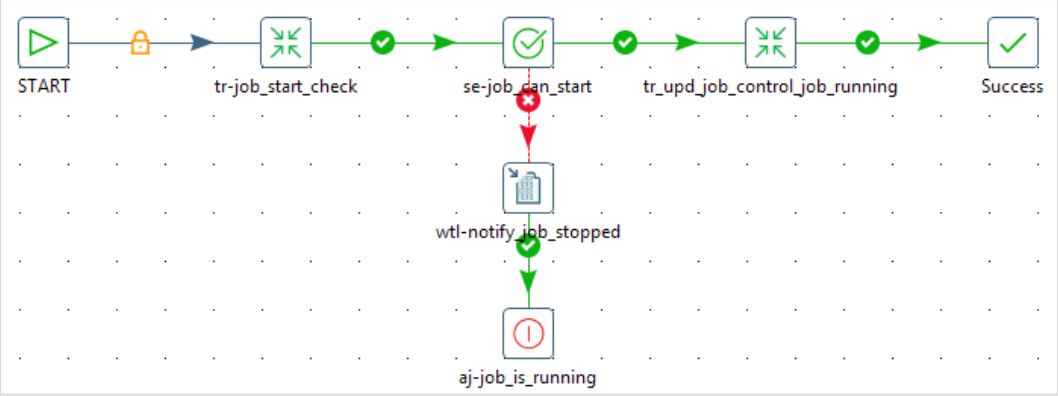

Figure 16: `jb_job_start_actions.kjb`

Finally, here is the `tr_job_start_check` transformation from within the `jb_job_start_actions` job:

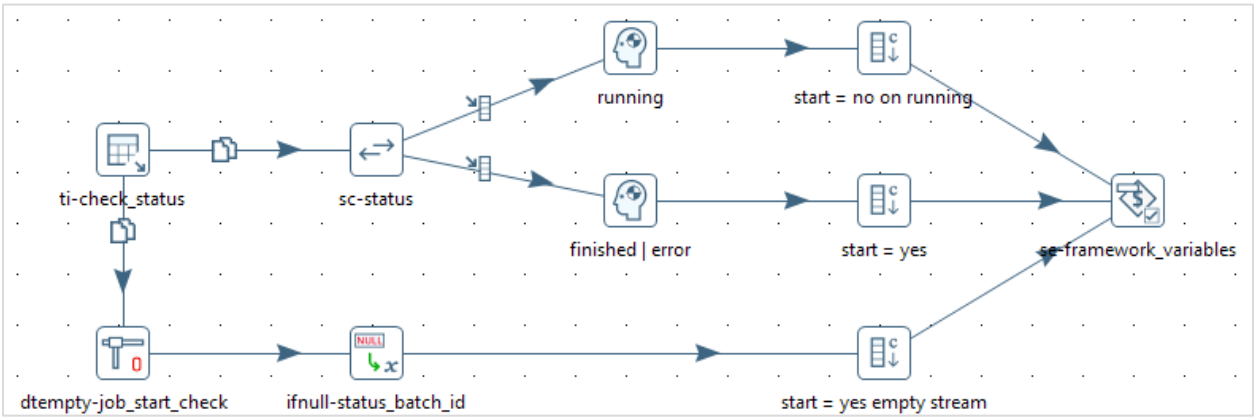

Figure 17: `tr_job_start_check`

The logic contained in the `jb_job_start_actions` job is the following:

> *IF last execution status `IS NULL` THEN start new job run*

As a result, three variables get populated, valid within the main job:

Table 22: Variables Set in `jb_job_start_actions`

Variable	Value
V_JOB_START_FLAG	Y
V_PREVIOUS_JOB_EXECUTION_STATUS	NULL
V_PREVIOUS_JOB_BATCH_ID	The batch_id retrieved from the job_control table

This only happens the first time a job runs. At this point there is no last execution status available in the `job_control` table.

> *IF last execution status = `'finished' OR 'error'` THEN start new job run*

Table 23: Variables Set if `'Finished'` or `'Error'`

Variable	Value
V_JOB_START_FLAG	Y
V_PREVIOUS_JOB_EXECUTION_STATUS	finished | error
V_PREVIOUS_JOB_BATCH_ID	The batch_id retrieved from the job_control table

> *IF last execution status = `'running'` THEN*

Table 24: Variables Set if `'Running'`

Variable	Value
V_JOB_START_FLAG	N
V_PREVIOUS_JOB_EXECUTION_STATUS	running
V_PREVIOUS_JOB_BATCH_ID	The batch_id retrieved from the job_control table

The main job will only start if `V_JOB_START_FLAG = 1`. As a result, the `job_control` table indicates this new status.

For example, for the `jb_main_sales_dwh_staging`, it will look like this (only displaying a subset of the columns):

Table 25: `job_control` Table Columns

Jobname	work_unit	status	project
jb_main_sales_dwh_staging	job-checkpoint	running	framework

Dependency Management

Dependencies in the main job are managed by having the work units executed in a specific sequence. The next work unit can only start if the previous was successful.

Job Restartability

Job failures should only require a simple restart of the main job. When a main job fails, the scheduler will only need to trigger a new execution for the framework to take care of restarting the job (more information on restarting is available in *Restartability in PDI* in the Pentaho Data Integration library). This comes down to the following logic. Remember the concepts that we discussed in the DI Framework introduction section:

- **Main job:** The job that is being scheduled
- **Work unit:** The jobs or transformations that are being executed in sequence by the main job

Work units are checkpoints at which restartability gets implemented:

- If the previous main job executed successfully (`V_PREVIOUS_JOB_EXECUTION_STATUS = 'finished'`), all work units will execute normally.
- If the previous job failed (`V_PREVIOUS_JOB_EXECUTION_STATUS = 'error'`), you skip the work units that executed successfully, rerun the work unit that failed, and have a normal execution for those work units that did not yet get executed in the previous run.

A work unit could be a transformation:

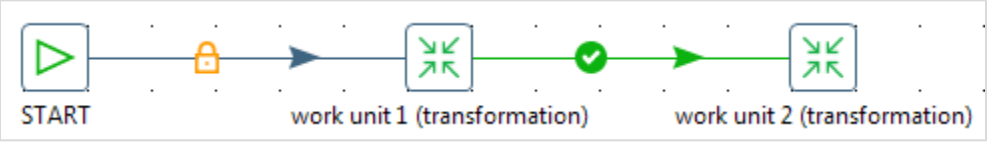

Figure 18: Transformation Work Unit

A work unit could also be a job that executes one or more jobs or transformations, like this:

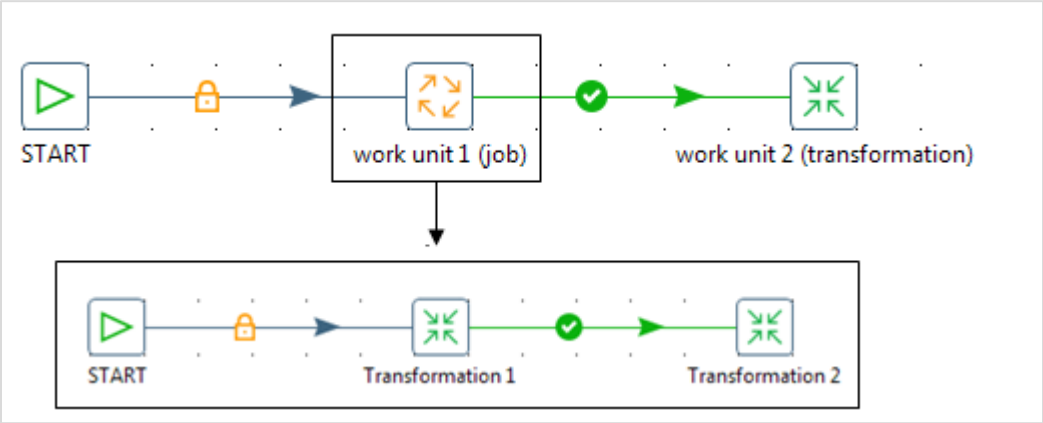

Figure 19: Job Work Unit

In both examples, restartability only gets implemented at the work unit level:

- If the main job fails because work unit 1 fails, in the next main job run, work unit 1 will be restarted and work unit 2 will have a normal run.
- If the main job fails because work unit 2 fails, in the next main job run, work unit 1 will not execute because it already executed successfully and work unit 2 will be restarted.
- If the main job executed successfully, in the next main job run, both units will have a normal run.
- The work unit can also be configured to start independent of the previous status.

You saw in the previous section that the `jb_job_start_check` job checks the last execution status of the job before it decides if it can start again or not. This check populates the following variables:

Table 26: jb_job_start_check Variables

Variable	Value
V_JOB_START_FLAG	`Y | N`
V_PREVIOUS_JOB_EXECUTION_STATUS	`finished | error | running`
V_PREVIOUS_JOB_BATCH_ID	The `batch_id` retrieved from the `job_control` table

In a similar fashion, you also have a **work unit start check** that gets executed before the execution of every work unit. Before we explain the work unit start check, let's have a look at how you keep track of the status of main job and its work units in the **job_control** table.

The **main job** can have one of three statuses:

Table 27: Main Job Status

Status	Description
`running`	The main job is still running.
`finished`	All work units finished successfully, so the complete main job finished successfully.
`error`	A particular work unit failed, so the complete main job failed.

A work unit can have one of four statuses:

Table 28: Work Unit Status

Status	Description
`running`	The work unit is still running.
`success`	The work unit completed successfully, but the main job is still executing other work units.
`error`	The work unit failed and also made the job fail.
`finished`	This work unit together with all other work units finished successfully, and therefore the complete main job finished successfully.

Notice the important difference between `success` and `finished`. `Success` means you are only partially done, while `finished` means the complete job finished successfully.

Let's focus on the first example of a work unit, where it is just a transformation. In it, the main job executes two work units, both transformations: work unit 1 and work unit 2.

If work unit 1 failed, `job_control` will look like this:

Table 29: `job_control` for Work Unit 1 Failure

jobname	work_unit	status
main job 1	job-checkpoint	error
main job 1	work unit 1	error
main job 1	work unit 2	NULL

Notice you have a separate `job_control` record to keep track of the main job status. This record is indicated by `work_unit = 'job-checkpoint'`. All other records keep track of the work unit status.

If work unit 2 failed, `job_control` will look like this:

Table 30: `job_control` for Work Unit 2 Failure

jobname	work_unit	status
main job 1	job-checkpoint	error
main job 1	work unit 1	success
main job 1	work unit 2	error

If both work units execute successfully, it will look like this:

Table 31: `job_control` for Successful Work Unit Execution

jobname	work_unit	status
main job 1	job-checkpoint	finished
main job 1	work unit 1	finished
main job 1	work unit 2	finished

Returning to the work unit start check that gets executed before the execution of every work unit: This functionality is implemented by the `tr_work_unit_start_check` transformation (`framework/control/tr_work_unit_start_check.ktr`).

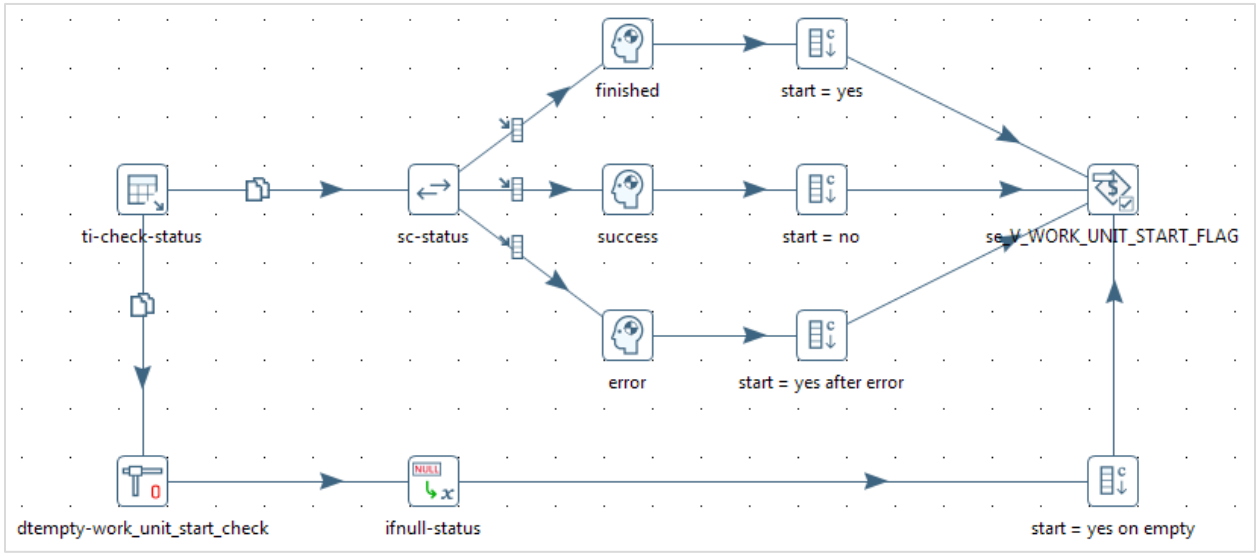

Figure 20: `tr_work_unit_start_check` Transformation

The logic contained in the `tr_work_unit_start_check` transformation is the following:

> *IF last execution status `IS NULL` THEN start new work unit run*

This only happens the first time a work unit runs. At this point there is no last execution status available. As a result, two variables get populated, valid within the work unit:

Table 32: Variable Population with Last Execution Status `Null`

Variable	Value
V_WORK_UNIT_START_FLAG	Y
V_PREVIOUS_WORK_UNIT_EXECUTION_STATUS	NULL

> *IF last execution status = `'finished' OR 'error'` THEN start new work unit run*

This happens when the complete main job finished successfully or errored. In case of main job error, it was this work unit that made the job fail.

Table 33: Variable Population with Last Execution Status `'Finished'` or `'Error'`

Variable	Value
V_WORK_UNIT_START_FLAG	Y
V_PREVIOUS_WORK_UNIT_EXECUTION_STATUS	`finished | error`

> *IF last execution status = `'success'` THEN do not start the work unit*

This happens when the complete main job did not finish successfully, but this work unit did. Therefore, in the next run, this work unit can be skipped.

Table 34: Variable Population with Last Execution Status `'Success'`

Variable	Value
V_WORK_UNIT_START_FLAG	N
V_PREVIOUS_WORK_UNIT_EXECUTION_STATUS	success

The work unit has one special mode of operation. When `P_CHECK_START = 'N'`, you will always execute the work unit, independent of its previous execution status.

Data Restartability

When the main job fails at a certain work unit, the failed work unit might already have processed some data before it failed. For that work unit to be restarted as part of the job restartability functionality, it has to clean up the previous execution before rerunning the work unit, by deleting the previous loaded records from the output table or removing output files from an output directory.

Because of both the **job and work unit start check**, the developer working on the actual work unit part of the DI solution has the following variables available:

Table 35: Variables from Job and Work Unit Start Checks

Variable	Value
V_JOB_START_FLAG	Y | N
V_PREVIOUS_JOB_EXECUTION_STATUS	NULL | finished | error | running
V_PREVIOUS_JOB_BATCH_ID	The batch_id retrieved from the job_control table
V_WORK_UNIT_START_FLAG	Y | N
V_PREVIOUS_WORK_UNIT_EXECUTION_STATUS	NULL | success | finished | error

The framework does not provide any data restartability itself, but makes these variables available so the developer can use them to decide if a cleanup is needed before the work unit can run.

You would probably just take the `V_PREVIOUS_WORK_UNIT_EXECUTION_STATUS` into account. Based on that value, you might need to clean up the records previously processed and identified in the tables by the `V_PREVIOUS_JOB_BATCH_ID` value in the `batch_id` column, if this value was loaded in the target tables.

Depending on the design of the work unit, this **cleanup** looks different:

- If the work unit is a transformation or job that was made database transactional, no cleanup would be needed and a rerun simply reprocesses the same data.
- If this was not the case, it might be that the cleanup involves deleting previously loaded records before starting the load in case an error happened.

When you make a job or transformation database transactional (rollback), changes to a data source occur only if a transformation or job completes successfully. This provides you with automated rollback behavior.

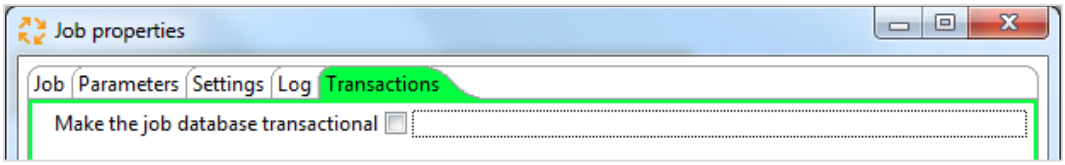

*Figure 21: **Make the job database transactional** Checkbox*

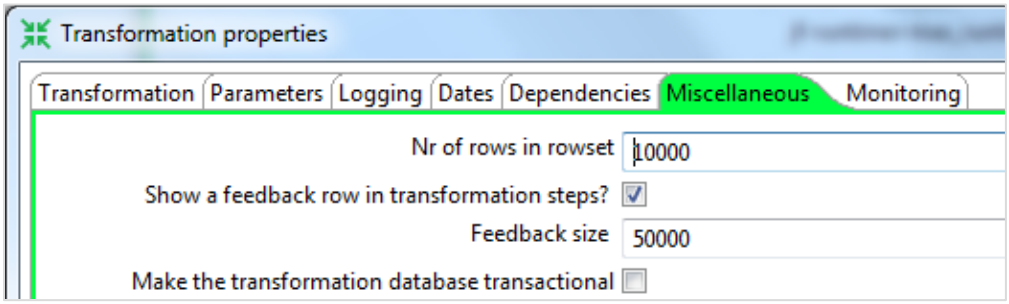

*Figure 22: **Make the transformation database transactional** Checkbox*

Transformation and Job Error Handling

There could potentially be many things in a job or transformation that can go wrong, some of which you are in control of, others of which are outside your control.

When you can catch the error, you can write specific messages to the log or abort the job or transformation when needed. Most steps in a transformation support error handling. You can do proactive checks (such as, does the file exist?), you can implement conditional logic (such as success or error hops in a job, filter logic in a transformation), you can send error mails, and so forth.

When you cannot catch the error (for example, when the server goes down), you can always rely on the job and data restartability functionality to clean up the faulty data and rerun the job.

The following error handling options are available for you in PDI:

- **Write to log** job entry or transformation step: This allows you to write specific error messages to the log. This can be important for the integration with external monitoring applications.
- **Abort** job entry or transformation step: This aborts the job or transformation based on a specific logic.
- **Transformation step error handling** can write faulty records to a different stream to handle them separately if needed.
- **Main job error email**: This writes an email to the admin team when something goes wrong and needs their attention.

Proactive checks in a job can be used where needed:

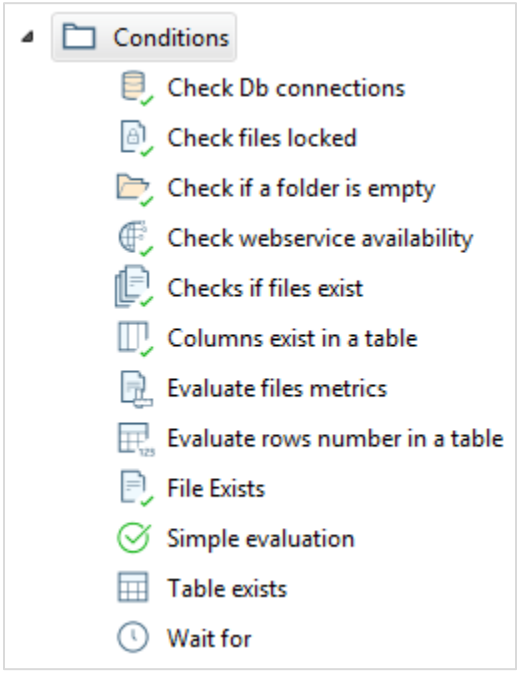

Figure 23: Job Conditions

The framework uses these options all the time. Each main job that is run generates its individual job log file, and those generated job run log files are saved in the same log file directory. So that debugging can happen in an effective manner, all log messages (write to log job entry and transformation step, abort job entry or transformation step) follow the same message format:

```
*** ERROR: <DETAILS> ***
```

These are actual errors that happened and caused the job to fail. If this happened, you caught the error appropriately and logged the error details accordingly

In these cases, `<DETAILS>` holds all the information needed to make a good log subject. When you are using variables or parameters, include their name as well for ease of reference. For example: `P_JOB_NAME = ${P_JOB_NAME}`. Sometimes, no variables or parameters are available, so the details will be hardcoded.

```
*** INFO: <DETAILS> ***
```

These are informational messages written to the log for debugging purposes.

```
*** WARNING: <DETAILS> ***
```

Warning info messages do not cause the ETL to fail, but prompt for action.

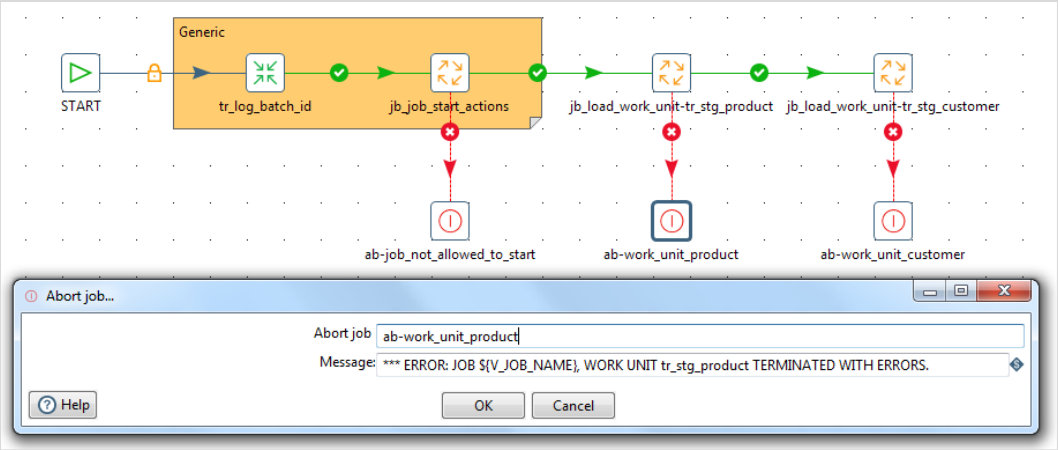

Figure 24: Abort Job Message Setting

It is also possible to have the `tr_upd_job_control_job` job (`framework/content-pdi/control/tr_upd_job_control_job.ktr`) send out an **error mail** when the main job errors. This behavior can be configured with the following `framework.properties`:

Table 36: `framework.properties` for Error Mail

Variable	Description

`SMTP_SERVER Connection Variables`	`SMTP_SERVER.HOST`
	`SMTP_SERVER.PORT`
	`SMTP_SERVER.USER`
	`SMTP_SERVER.PASS`

`ERROR_MAIL Variables`	`ERROR_MAIL.SEND_FLAG=0 | 1`
	`ERROR_MAIL.SENDER.ADDRESS=support@etl.com`
	`ERROR_MAIL.SENDER.NAME=ETL Admin`
	`ERROR_MAIL.DESTINATION.ADDRESS=`
	`ERROR_MAIL.SUBJECT_PREFIX=ETL ERROR`

For example, if the `jb_main_sales_dwh_staging` job experiences an error during execution:

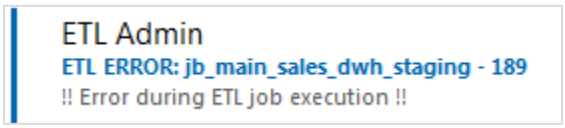

Figure 25: ETL ERROR Message

Then setting `ERROR_MAIL.SEND_FLAG = 1` would send the following error mail:

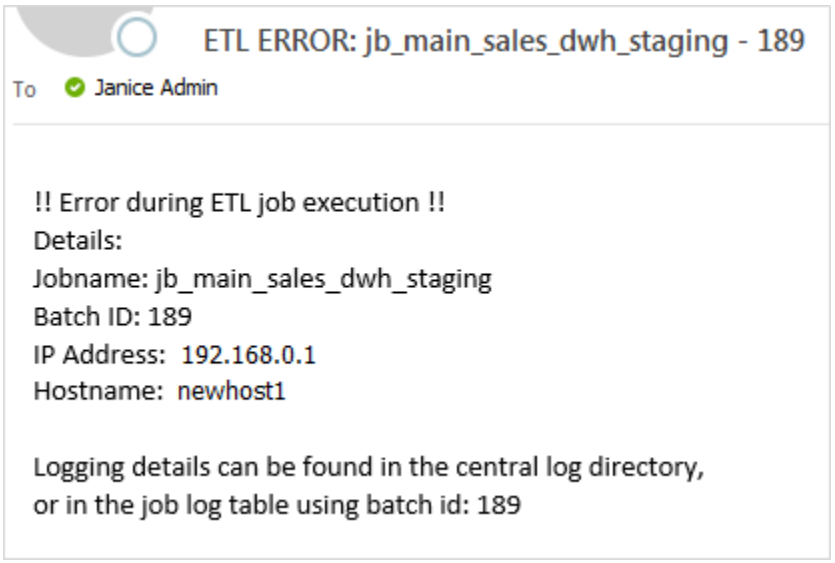

Figure 26: Error Mail Example

Launching DI Solution Work Units with jb_work_unit Job

One of the last framework artifacts that needs an introduction is the `jb_work_unit` job (`framework/content-pdi/execution/jb_work_unit.kjb`) that acts as a wrapper to launch the work units of the actual DI solution in a controlled manner.

This job builds on Diethard Steiner's example in his [Restartable Jobs blog post](#).

Use the `jb_work_unit` job to call the actual work units within the main job in sequence. Consider the following `jb_main_sales_dwh_staging` job:

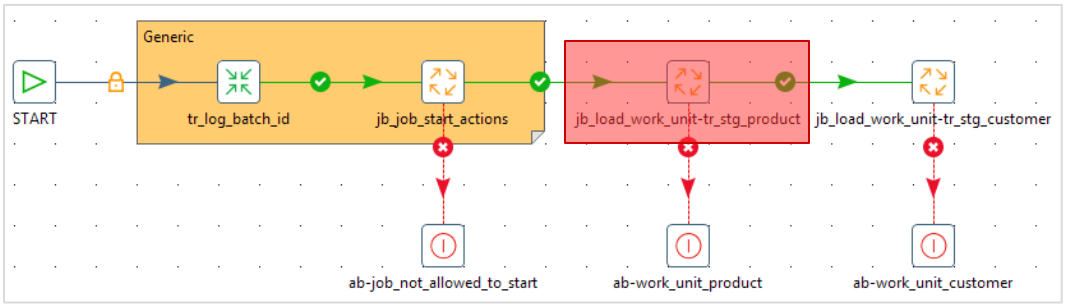

Figure 27: Main Job Calling `jb_work_unit` Job

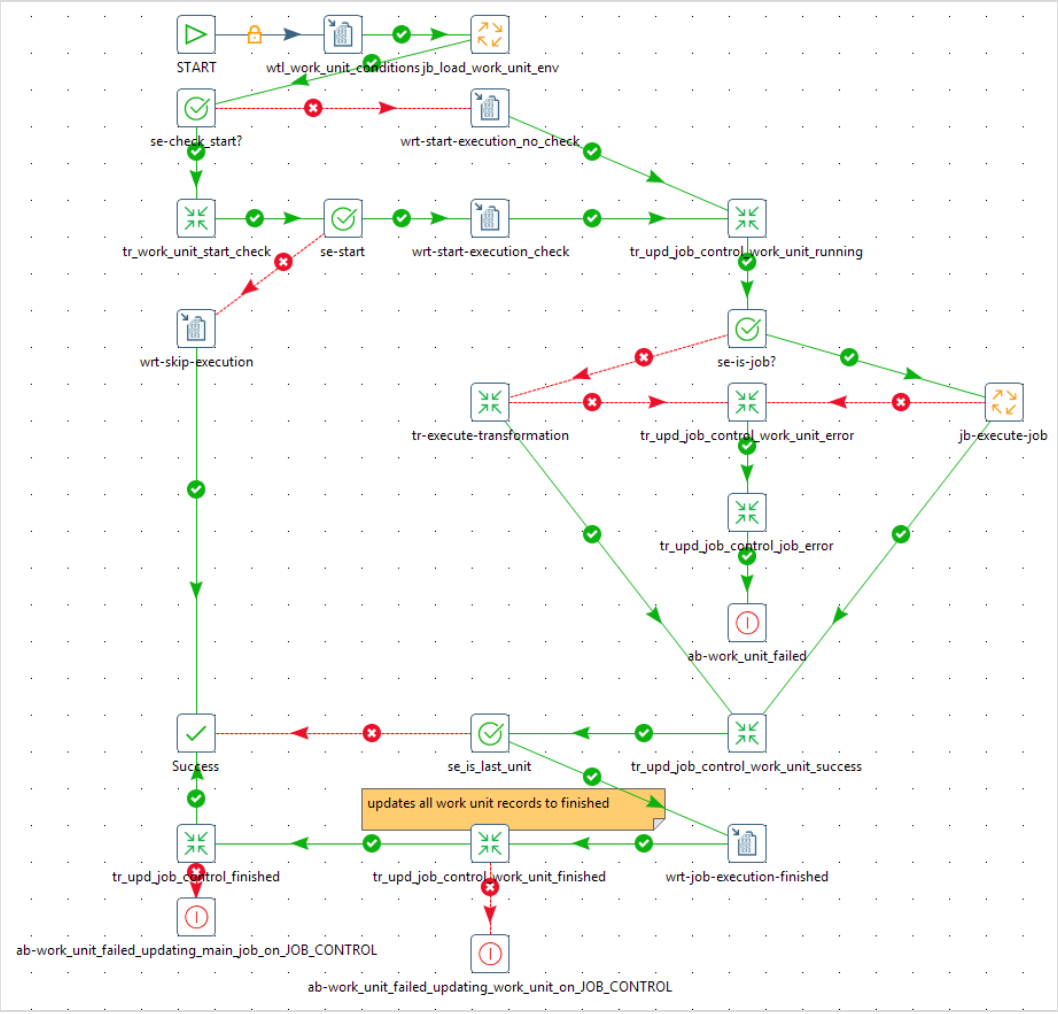

Figure 28: `jb_work_unit` Job

The `jb_work_unit` job orchestrates the execution of a single work unit. It accepts the following **input parameters**:

Table 37: `jb_work_unit` Job Input Parameters

Parameter	Description	Example
`P_CHECK_START`	`Y|N`, controls if the framework should perform the work unit start check	`Y` (this is the default if no values gets provided)
`P_PROJECT_NAME`	Name of the project the job is part of	`sales_dwh`
`P_WORK_UNIT_NAME`	File name of the transformation or job to be executed	`tr_stg_customer`
`P_WORK_UNIT_PATH`	Subfolder structure from `CONTENT_HOME`	`staging`
`P_WORK_UNIT_TYPE`	`job | transformation`	`transformation`
`P_IS_LAST_UNIT`	`Y|N` last work unit on job	`N`

The job then functions in this way:

1. Correct environment variables for the work unit's execution are then loaded through `jb_load_work_unit_env`, and after that, the job checks to see if it needs to perform a start check using `P_CHECK_START`.
2. If a start check needs to be performed, the job checks if the work unit can be started using the `tr_work_unit_start_check` transformation (`framework/content-pdi/control/tr_work_unit_start_check.ktr`). This can lead to three actions, dependent on the last execution status of the work unit:
    a. `finished`: work unit will have a normal execution
    b. `success`: work unit will be skipped
    c. `error`: work unit will be rerun
3. When the work unit can start, you register the work unit execution start in the `job_control` table with the `tr_upd_job_control_work_unit` transformation (`framework/content-pdi/control/tr_upd_job_control_work_unit.ktr`).
4. Execute the actual work unit job or transformation (`P_WORK_UNIT_NAME`, `P_WORK_UNIT_TYPE`).
5. Register the work unit execution status (`success or error`) with the `tr_upd_job_control_work_unit` transformation.
6. When the work unit fails, the main job stops and its status gets updated to `error` by the `tr_upd_job_control_job` job (`framework/content-pdi/control/tr_upd_job_control_job.ktr`).
7. When the work unit succeeds, launch the next work unit in the main job.
8. When the last work unit (`P_IS_LAST_UNIT = Y`) executes successfully, mark the main job and its work units as `finished` in the `job_control table` with the `tr_upd_job_control_work_unit` and `tr_upd_job_control_job` transformations.

Using the DI Framework

You have seen the main functionality of the DI framework, including:

- It will take care of a controlled execution of a main job, a workflow of work units (jobs or transformations) in sequence.
    - When a work unit fails, the complete jobs fails.
    - Regular executions happen if the previous load was successful (indicated by the `'finished'` status) or if this is the first time that this job gets executed (no record is available yet in the `job_control` table).
- It will restart the main job.
    - This happens if the previous run of the job errored (`status = 'error'`).
    - If work units executed successfully in the previous run, they will be skipped in this run.
    - If a work unit failed in the previous run, it will be restarted. The framework makes the previous execution status of the work unit available to the work unit through the `V_PREVIOUS_WORK_UNIT_EXECUTION_STATUS` (`'error'`). It is up to the actual work unit that gets executed to implement the actual data restartability.
- Work units can be a single transformation or a job that groups together other jobs and transformations.
    - Restartability is at the level of the work unit.
    - When your work unit is a job, restartability is at the level of that job in its entirety. The complete job needs to execute successfully for the work unit to be successful. When the previous execution errors, the complete job will be restarted.
- Executing work units in parallel is not possible.
    - When such behavior is needed, this can be dealt with by the scheduler: scheduling multiple jobs in parallel.

Now, let's look at how you put this framework to work in your project. When the DI development team starts to develop the actual DI solution, they will develop the actual work units and the main jobs that execute those work units. When working on those artifacts, they need to integrate with the DI framework and there are some specific rules to be followed for this to happen smoothly. This section describes those specific **integration points.**

You can find more information on these topics in the following sections:

- Setting Up the Local Environment
- Loading the Development Environment
- Referencing Other DI Artifacts
- Creating a Main Job
- Writing to Log Format
- Optimizing Database Logging
- Debugging
- Changing Execution to Local Filesystem or Pentaho Repository

Setting Up the Local Environment

Your project content and configuration are stored in Git. In order to set up your local development environment, you need to have access to a database(s) (local or remote) to hold the `pdi_control`, `pdi_logging`, and `sales_dwh` data. The create statements for the `pdi_control` and `pdi_logging` tables can be found in `framework/sql/framework.sql`.

Then, follow these steps:

1. Create the **project folder** (for example, `C:\pentaho\projects`) on your local filesystem.
2. **Clone** the following Git repositories to your local project folder:
    - `sales_dwh`
    - `sales_dwh-configuration`
    - `framework`
3. The tables for `sales_dwh` are automatically created by the solution. Check that you have these tables per connection:
    - `pdi_control`
        - `job_control`
    - `pdi_logging`
        - `log_channel`
        - `log_job`
        - `log_tran`
    - `sales_dwh`
        - `stg_customer`
        - `stg_product`
4. Make the following changes to the `pdi-config-local` environment configurations:

The `config-pdi-local` configuration will be different for each developer, so this configuration will never be committed to Git. A master version that developers can use as a starting point will be available.

    - `env.conf` (Windows or Linux version)
        - `PENTAHO_HOME`: Point this to your local Pentaho installation folder.
        - `KETTLE_CLIENT_DIR`: Optional. Leave blank or point this to your local Kettle client installation folder if this is different from `${PENTAHO_HOME}/design-tools/data-integration`.
        - `PENTAHO_JAVA_HOME`: Point this to your local Java installation folder.
        - `PENTAHO_DI_JAVA_OPTIONS`: Set the correct Java memory settings.
    - `.kettle/kettle.properties`
        - All variables values that contain directories must be correctly configured.
        - `PDI_LOGGING` connection properties must be set up.
    - `.kettle/shared.xml`
        - Connection definitions if a different database than PostgreSQL is used.
    - `properties/framework.properties`
        - `PDI_CONTROL` connection properties
    - `properties/sales_dwh.properties`
        - `SALES_DWH` connection properties

Loading the Development Environment

DI developers need to start the right **Spoon** for the right environment, avoiding using incorrect variables. Starting development activities on the local environment means starting the `sales_dwh-configuration/config-pdi-local/spoon.bat` or `spoon.sh` file.

Since only global variables are part of the `kettle.properties` file, and project and job variables are introduced in the `project` and `job.properties`, these variables are not by default available to Spoon. Therefore, the moment that you open Spoon, you must run the `jb_set_dev_env` job (`/framework/content-pdi/developer-tools/jb_set_dev_env.kjb`).

This job accepts the following variables:

Table 38: Variables for `jb_set_dev_env` Job

Parameter	Description
P_PROJECT	Name of the project
P_JOB_NAME	Name of the job (no extension needed)
P_WORK_UNIT_NAME	Name of the work unit (no extension needed)

Depending on the variable values, this job will load the framework, common, project, job (if it exists), and work unit properties (if they exist), and initiate the `V_JOB_NAME` variable.

Referencing Other DI Artifacts

When developing work units (job or transformation) you might need to reference other artifacts.

If this is the case, do not use the `${Internal.Entry.Current.Directory}` variable, but use the following variables made available by the framework. When working on jobs or transformations of a specific project (`sales_dwh`, `data_export`), you have variables available that point to a specific location within the project folder (such as `sales_dwh`).

Table 39: Variables to Reference Other DI Artifacts

Variable	Description
${CONTENT_HOME}	`sales_dwh/content-pdi`
${SQL_HOME}	`sales_dwh/sql`
${FILE_MGMT_HOME}	`sales_dwh/file-mgmt`

Creating a Main Job

Remember that the main job orchestrates the execution of the work units. For this to happen in combination with the framework, the main job needs to follow a certain template.

The `jb_main_job_template` job is available at `framework/content-pdi/developer-tools/jb_main_job_template.kjb`.

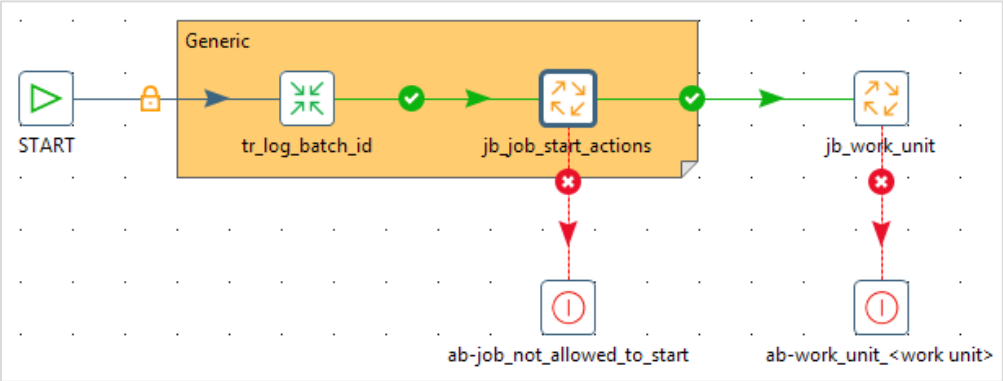

Figure 29: `jb_main_job_template` Job

Remember to always start your main job with the `tr_log_batch_id` transformation and the `jb_job_start_actions` job.

Writing to Log Format

See the Exception Handling section to understand the format that must be used for writing a message to the log and how to do this.

Optimizing Database Logging

Database logging is done automatically since the framework uses the `kettle.properties` database logging variables to define this at a global level. However, you can optimize the database logging for the transformations by adding the Step Name configuration for the `LINES_` metrics.

Debugging

We have talked about database logging being part of the framework. Next to this, the framework also writes all execution details of a single job run to a logging file in its `LOG_HOME` folder.

This log file has the following filename format: `${LOG_HOME}/${P_JOB_NAME}_yyyyMMdd_HHmmss.log`, so for example: `jb_main_sales_dwh_staging_20180705_104735.log`.

This file contains the PDI logging defaults based on the logging level selected, together with all messages written to the log with the framework or by using the **Write to log** and **Abort** entries or steps.

When your job or work unit halts for unexpected reasons, consult this logging file first either by checking the logging tab in Spoon, or by consulting the file directly.

Also, note that the `job_control` is updated with the job and work unit status. When testing locally, you can play with the status in that table to influence the job and work unit behavior.

You can also delete all records in the `job_control` table since they would be recreated when the job cannot find the records.

Changing Execution to Local Filesystem or Pentaho Repository

The framework allows for a local execution with Spoon, working file-based, and a Pentaho Server execution using the Pentaho Repository. Through the different framework variables available, it even allows for a different repository structure in Git (file-based) versus the Pentaho Repository.

Imagine the following setup: you have the same content, both file-based and in the Pentaho Repository, but a different structure is used.

File-based in the local environment: `C:\pentaho\projects`

```
|-- sales_dwh-configuration
|     |-- config-pdi-local
|-- sales_dwh
|     |-- content-pdi
|     |     |-- tr_my_ktr_sales_dwh.ktr
```

The resulting variables to support this setup would look like this:

Table 40: Variables for File-Based Solution in Local Environment

File	Variable	Description
kettle.properties	**ROOT_DIR**	C:/pentaho/projects
	CONTENT_DIR	C:/pentaho/projects/
	CONFIG_DIR	${ROOT_DIR}
	PROJECT_ENV	config-pdi-local
sales_dwh.properties	**CONTENT_HOME**	${CONTENT_DIR}/sales_dwh/content-pdi

Pentaho Repository in the dev environment: `/public/`

```
|-- sales_dwh
|     |-- tr_my_ktr_sales_dwh.ktr
```

The following variables will solve this change in structure by abstracting the location:

Table 41: Variables for Pentaho Repository in the Development Environment

File	Variable	Description
kettle.properties	**ROOT_DIR**	C:/pentaho/projects *This always points to a location on the filesystem since only the content-pdi part of the projects will move to the Pentaho Repository. Things like configuration or sales_dwh/file-mgmt always stay on the filesystem.*  *This variable, included for completeness, is actually not part of the kettle.properties file and is set in the framework/bin/init.sh/bat script.*
	CONTENT_DIR	/public
	CONFIG_DIR	${ROOT_DIR}
	PROJECT_ENV	config-pdi-dev
sales_dwh.properties	**CONTENT_HOME**	${CONTENT_DIR}/sales_dwh

As a result, the solution can always reference the tr_my_transformation_sales_dwh.ktr transformation in the same way: ${CONTENT_HOME}/tr_my_transformation_sales_dwh.ktr.

Documenting Your Solution

ETL documentation to help solution development and support should be kept under the corresponding project folder in Git (such as sales_dwh/documentation). We recommend that you write the documentation in Markdown notation to make it light and easy to read.

You should version your documentation the same way that you version your DI artifacts.

Developers should also add comments with notes in each job or transformation. These notes allow anyone reviewing the code, or taking over support of it, to understand decisions behind the logic or changes. Developers can apply different colors to the notes depending on the type of note.

Automating Deployment

As your project moves through its lifecycle, the deployment process will make sure all necessary project artifacts become available on the required Pentaho Server machines.

In a situation such as our example, when you deploy a test or production release to the Pentaho Server machines, the Git repositories will be checked out to the local filesystem of these machines. Only the jobs and transformations from the etl repository end up in the Pentaho Repository; all other artifacts stay on the filesystem so that they can be referenced from the jobs and transformations at execution time.

For example, a job might reference a shell script to assist the job in executing a specific task. This script will be available from the filesystem and not the Pentaho Repository.

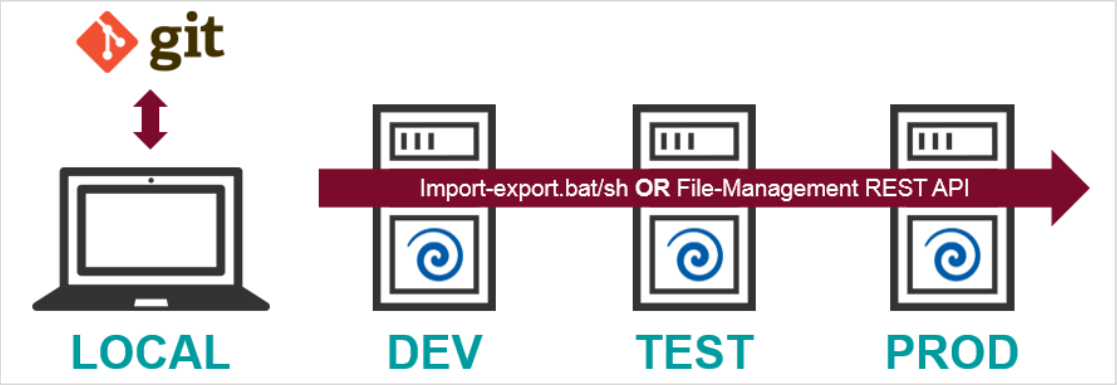

Figure 30: Example Basic Automation Flow

The PDI toolkit offers multiple functionalities that can support and automate your deployment process.

You can upload content to the Pentaho Repository from the command line using the `import-export.bat/sh` script available in the `pentaho-server` directory. See the Command Line Arguments Reference to get a complete overview of all command line arguments available.

An example of a deployment using this script is available at `configuration/deploy-pdi/deploy.bat`. This script deploys a `<project>.zip` file (all content from the `content-pdi` folder zipped) and asks for the target environment. It uses the `DI_SERVER_` variables from the configuration files (`config-pdi-<env>`) of the target environment to get the Pentaho Server credentials to perform the upload.

```
Enter Project Name: sales_dwh
Enter Environment (dev/test/prod): dev
IMPORTING sales_dwh into Environment dev using installation in C:\Pentaho810
Environment variable ################################################## not defined
Environment variable #              CORE PROJECT VARIABLES             # not defined
Environment variable #  Seperate variables for maximum flexibility     # not defined
Environment variable ################################################## not defined
Environment variable ################################################## not defined
Environment variable #              FRAMEWORK VARIABLES                # not defined
Environment variable ################################################## not defined
Environment variable ################################################## not defined
Environment variable #                SERVER VARIABLES                 # not defined
Environment variable ################################################## not defined
Environment variable #BA_SERVER not defined
Environment variable #DI_SERVER not defined
Environment variable ################################################## not defined
Environment variable #                  LOGGING SETUP                  # not defined
Environment variable ################################################## not defined
Environment variable # LOGGING CONNECTION not defined
Environment variable # LOGGING KETTLE VARIABLES not defined
DEBUG: Using PENTAHO_JAVA_HOME
DEBUG: _PENTAHO_JAVA_HOME=C:\Pentaho810\java
DEBUG: _PENTAHO_JAVA=C:\Pentaho810\java\bin\java.exe
DEBUG: PENTAHO_INSTALLED_LICENSE_PATH=C:\Pentaho810\.installedLicenses.xml
Picked up JAVA_TOOL_OPTIONS: -Djava.vendor="Sun Microsystems, Inc"
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
Import was successful
Press any key to continue . . .
```

Figure 31: Example Output from `deploy.bat` Script

Another possibility for automating the deployment is using the Pentaho Server's File Management APIs**.** The `tr_deploy.ktr` transformation (`configuration/deploy-pdi/tr_deploy.ktr`) shows an example of deploying using the REST API with PDI.

Once you deploy the project jobs and transformations to the Pentaho Repository, the `shared.xml` file will no longer be used to hold the database connections. Instead, the connection information will then be stored in the Pentaho Repository.

The deployment process does not use the information from the `shared.xml` file, but it uses the connection information that is stored as a backup within the transformation and job's `XML` definition.

Once a connection is created within the Pentaho Repository, it does not get overwritten automatically by the automated deployment. if you change the project's connection details, you will need to update the Pentaho Repository connection manually or use the REST API to do so.

Creating a Development Guidelines Handbook

When you are working on a DI project with multiple developers collaborating on the final DI solution, we recommend creating a Development Guidelines handbook.

This handbook should include guidelines and standards that must be followed by the entire team when developing the solution, and that can be implemented in a set of QA checks on the final solution.

The handbook must also include naming standards. The framework also follows naming conventions for its job and transformation names and job entry and step names based on *Naming Standards for PDI* in the PDI document library.

DI Framework Demo

Configure and run the **demo** to see the framework in action using a sample project. Configuring the demo is easy, and all steps are documented in the Using the DI Framework section.

Once configured, you can launch the `jb_main_sales_dwh_staging` job with the `jb_launcher` job (`framework/content-pdi/execution/jb_launcher.kjb`). Consult the `job_control` table to see what a successful execution looks like.

Test job restartability by triggering an error during execution:

1. Open the `tr_stg_customer` transformation (`sales_dwh/content-pdi/staging/tr_stg_customer.ktr`).
2. Enable the hop coming from the `customers-with-errors.csv` step.
3. Make sure to disable the other hop.
4. Run this once and see the result of the failing job in the `job_control` table.
5. Revert back to the `customer.csv` input step and run the job again with the launcher job.

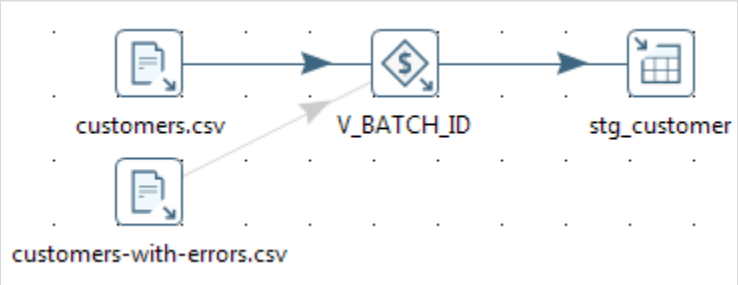

Figure 32: `tr_stg_customer.ktr`

Extending the Current DI Framework

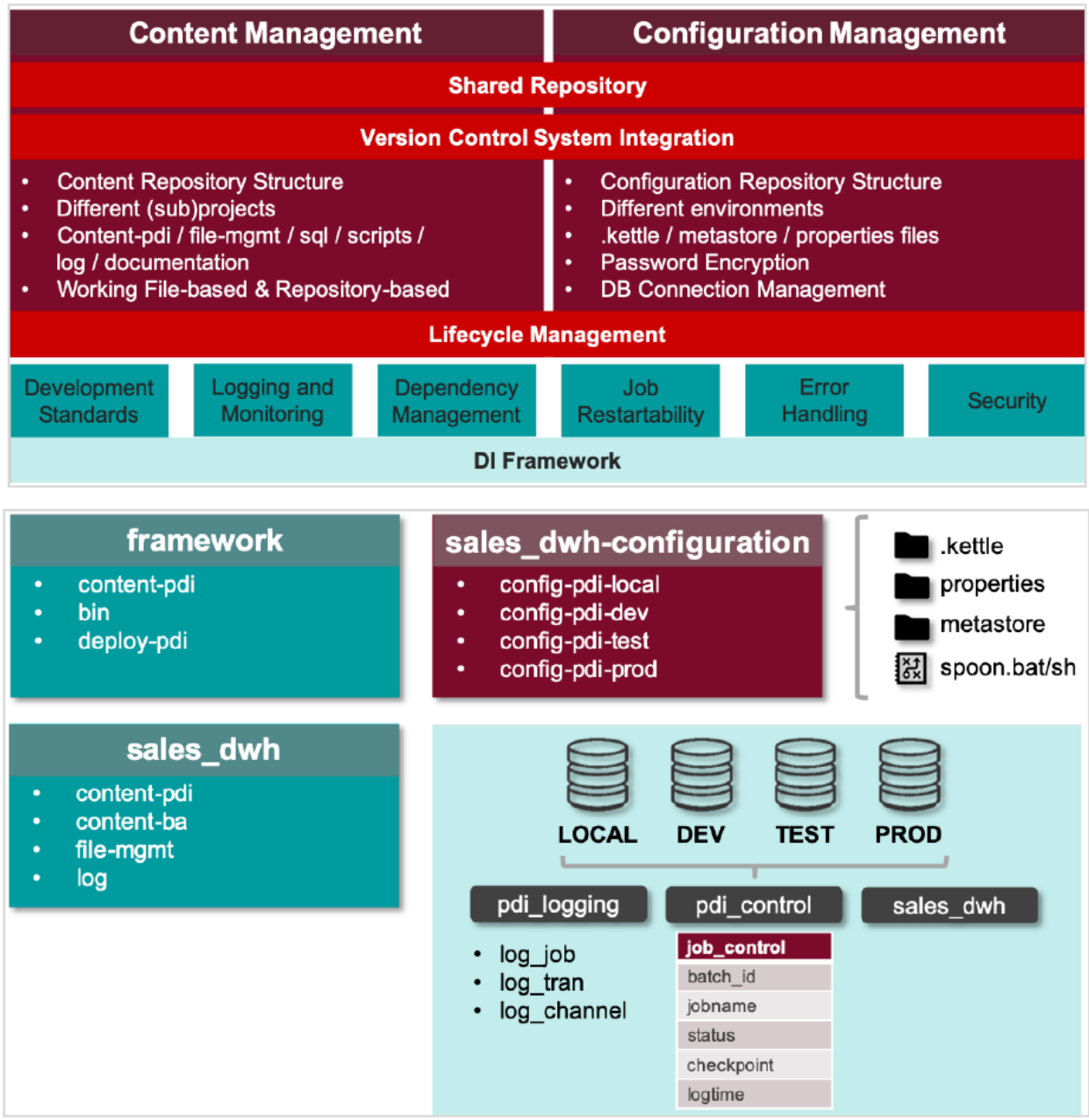

Figure 33: Project Setup Summary

It is important to remember that the DI framework suggested in this document only provides a starting point for your DI projects. The realities in your project might require you to make changes to the framework's current behavior or to extend its functionality in general.

You see there is no such thing as a one-size-fits-all for this. However, it should provide you with a good starting point, already considering multiple best practices that set your project up for success.

In your sample setup, you had two projects (`sales_dwh` and `data_export`) in different repositories. In general, it is only recommended to keep projects together in the same repository when they follow the same release cycle. The reason is that most Git functionalities (branching, tagging, creating releases, and so forth) work at the level of the repository. If you have two separate projects requiring their own lifecycles, you should set them up as separate repositories, following a similar structure as suggested within this document.

Apart from storing DI artifacts, this project setup could also be extended to hold BA (reports) and Server artifacts (database connections, schedules). This would allow the Pentaho Server to be set up from scratch using the artifacts stored in Git.

Shared Artifacts Between Projects

Assume the following situation: The `sales_dwh` and `data_export` projects need to have a home for artifacts that will be shared between both projects. One could argue that such content could be placed in the `framework` repository. However, this might be specific solution content that does not fit within the `framework` repository.

We can treat such content as a specific project called `common`, having its dedicated Git repository. We only need an `etl` repository, since the configuration can be part of the project's configuration repositories: all projects could use this `common` content in a slightly different way, configurable through a `common.properties` file.

To enable this for the `sales_dwh` project, add the following variables:

Table 42: Variables to Enable Shared Artifacts

File	Variable	Description
`kettle.properties`	`COMMON_DIR`	`C:/pentaho/projects` Root folder that the `common` repository will be part of. This always points to a location on the filesystem since only the `content-pdi` part of the projects will move to the Pentaho Repository.
`project.properties`	`COMMON_HOME`	`${COMMON_DIR/common/content-pdi` Variation on the `CONTENT_HOME` principle. You need a `COMMON_HOME` variable so other projects have a way of referencing content from the `common` project. When working in a specific project, `CONTENT_HOME` is your valid starting point and `COMMON_HOME` is how you reference content from the `common` project.
`common.properties`	To be defined (optional)	This depends on the artifacts within the `common` repository and how they need to be made dynamic.

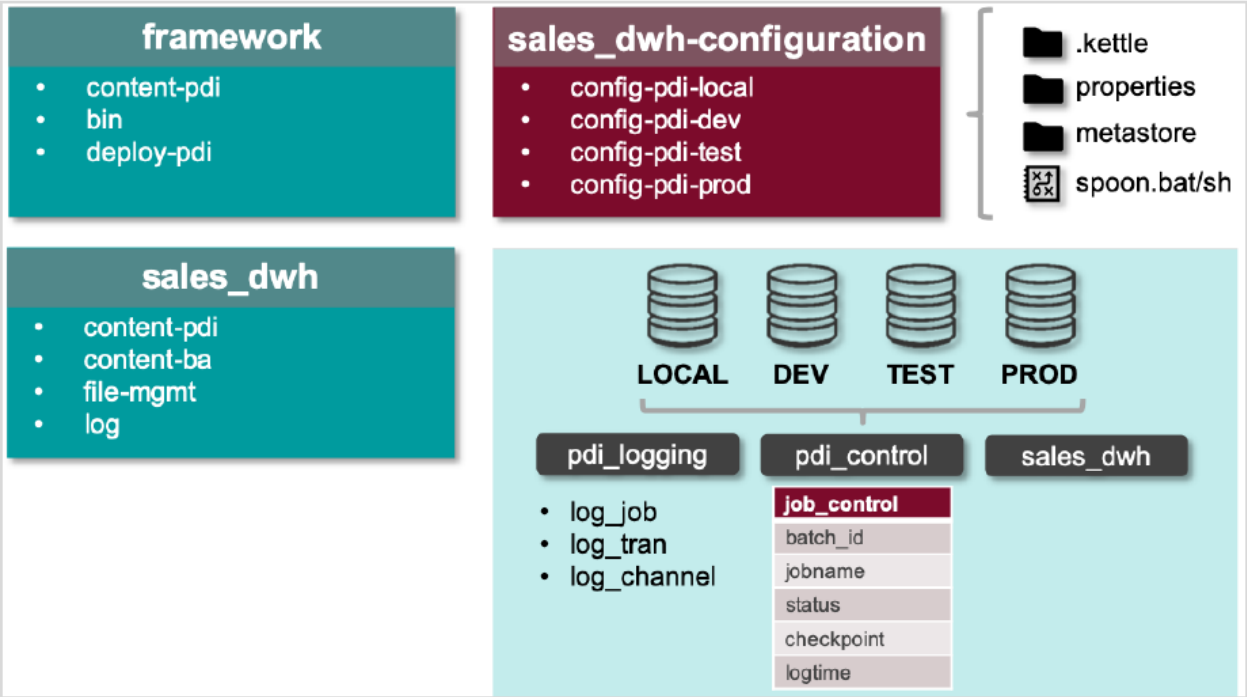

Figure 34: Project Setup Extension

Running Multiple Projects Within a Single Pentaho Server

Assume your Git setup is as discussed within the document: two projects that each have their dedicated `etl` and `configuration` repository. This makes sense at development time, since each project is being maintained by its own team, and, as such, each project has its own lifecycle.

However, you might run into a situation where, once past development, the projects need to be deployed into a single Pentaho Server. This conflicts with our current setup since all of our projects would have their own dedicated Pentaho Server, started through the `start-pentaho.sh/bat` script within the `config` folder for the right project environment.

In order to make this work, we need to introduce a new Git repository called `platform-configuration`. This repository will house all platform environment configurations, one per Pentaho Server environment, which we will call "platform" here. Make sure to name the `config` folders exactly the same as for your other projects.

Each environment can only have a single `.kettle` and `metastore` folder, so we need to integrate those form the different projects that will be running within the same Pentaho Server. This mostly impacts the `kettle.properties` files and the `shared.xml` file. For the `kettle.properties` file, this works fine since it only contains global variables that are mostly defining key project directories.

Apart from that, only the `content-pdi` folder of all the projects will be deployed to the Pentaho Server. All other content from the `etl` and `config` repositories will stay on the local filesystem. This way, whenever a job from a specific project is launched through the `jb_launcher` job (`framework/content-pdi/execution/jb_launcher.kjb`), it will load the project-specific

variables from its `properties` files from the `config` repository. As a result, we will have different jobs from different projects that can run at the same time, all using a variable called `CONTENT_HOME` which is pointing to a different project's content folder. This is not a problem, since those variables' scope is limited to their root job.

Related Information

Here are some links to information that you may find helpful while using this best practices document:

- Demo Files
- AES
- AES Security
- Command Line Arguments Reference
- Configuring Log Tables for Concurrent Access
- File Management APIs
- Markdown
- Nonfunctional Requirements (NFRs)
- Pentaho Components Reference
- Pentaho Data Integration Library
- Pentaho Installation
- Transactional Databases and Job Rollback
- Use Version History