



Continuous Integration (CI) with Pentaho Data Integration

Contents

- Overview..... 1
 - Before You Begin..... 1
 - Terms You Should Know..... 2
 - Demo Download Link..... 2
- Introduction to Continuous Integration 3
- Testing Strategy..... 4
- Continuous Integration Infrastructure 8
 - Test Environment..... 8
 - CI Server 8
 - Test Framework and Suite 9
 - Steps for Test Structure 9
 - Elements of Test Framework..... 9
 - Using PDI to Test PDI..... 10
- Continuous Integration with Pentaho 11
 - Project Foundations..... 12
 - Test Suite and Framework Definition 12
 - config-pdi-test Environment..... 13
 - Tests Folder..... 13
 - Test Case Example 15
 - Test Framework..... 15
- Configuring the CI Server Workflow 19
 - Demo Starting Point 19
 - Installing Jenkins 19
 - Configuring config-pdi-test..... 20
 - Jenkins Continuous Integration Server 20
 - Creating a Jenkins Project..... 20
 - Source Code System Definition 21
 - Trigger the Process..... 22
 - Building the Code and Performing Tests..... 23
 - Evaluate Testing Results 25
 - Solution Package Generation..... 28

CI Server Workflow Summary	29
Possible Extensions to CI Workflow Setup	31
Next Steps	31
Related Information.....	32

Overview

This document introduces the foundations of Continuous Integration (CI) for your Pentaho Data Integration (PDI) project. It is the third document in the [PDI DevOps series](#). This series consists of documents that provide guidance on creating an automated environment where iteratively building, testing, and releasing a PDI solution can be faster and more reliable, resulting in a high-quality solution that meets customer expectations at a functional and operational level.

Our intended audience is Pentaho administrators and developers, as well as IT professionals who help plan software development.

We are focusing on the building blocks which you can customize or extend to match your specific requirements:

- Testing strategies
- What is needed in terms of CI infrastructure and how to do CI with Pentaho
- What the required project foundations are for your CI strategy
- How to design a test framework and suite with PDI and how to integrate this with a CI server for automating the end-to-end CI workflow

The examples and instructions are geared toward a situation where you are using Git as a code repository, Jenkins as an automation server and Junit as the test framework, but other configurations would work if you applied the same principles found throughout this document.

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 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. Make sure to review the content discussed in the previous documents from the PDI DevOps series:

- [Driving PDI Project Success with DevOps](#)
- PDI Project Setup and Lifecycle Management (in the [PDI Best Practices library](#))

Terms You Should Know

Here are some terms you should be familiar with:

- **Versioning:** Creating a new version of a system or software is referred to as versioning. A version control system (VCS) is a method to keep records of changes to software over time so that you can avoid conflicts of different people saving over each other's changes and can also roll back the code to a previous version if you need.
- **Software Build:** The process of compiling computer source code into binary code, packaging binary code and running automated tests is referred to as a software build. In software development this is a necessary pre-condition for continuous integration and automated testing. Although no compiling is required for a PDI solution, this term is used interchangeably for the process of executing the scheduled tests for your PDI solution increment and packaging the increment when tests are executed successfully.
- **Fixture:** a data set used for testing.
- **Test suite:** collection of test cases.

Demo Download Link

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

Introduction to Continuous Integration

Continuous integration (CI) is the process of iteratively merging code changes made by the development team into the central solution repository and testing those changes as early and often as possible. The team's changes can then be validated by creating a solution build and running automated tests against the build, giving the developer immediate feedback and the ability to easily fix bugs. This allows you to avoid integration problems that can happen when people wait for release day to merge their changes into the central repository.

CI within Pentaho projects supports your Agile project delivery methodology and is important to the success and lifecycle of your data integration solution. Data integration solutions benefit from automated testing in the same way any other software does, by checking that the application is not broken whenever new iterations are integrated into the central solution repository. CI lets you focus more on development by acting as a second line of defense, letting you know of errors as soon as possible.

Continuous Integration (CI) is often the first step of a DevOps implementation in an organization and is the most common DevOps maturity phase achieved.

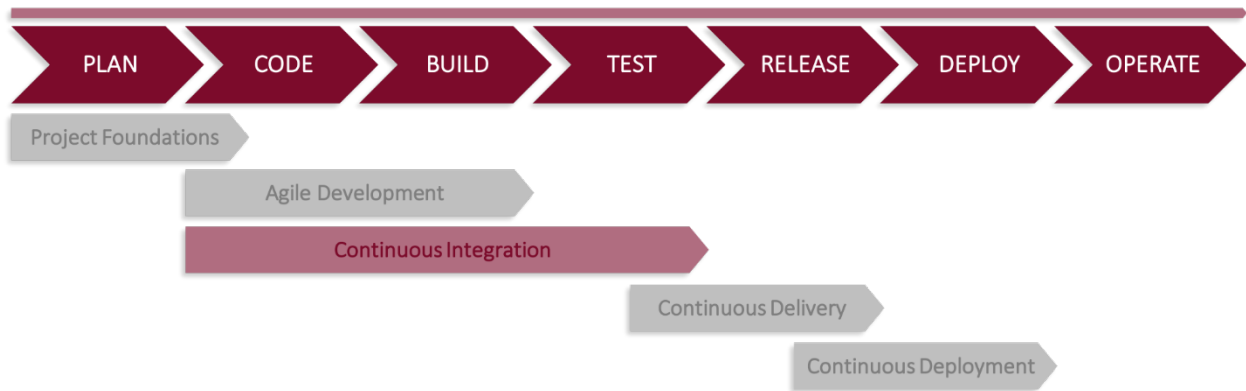


Figure 1: CI DevOps Maturity Phase

Testing Strategy

We introduced Continuous Integration as a DevOps maturity phase that has great emphasis on test automation to check that the solution is not broken whenever new iterations are integrated into the central solution repository. Effectively running automated tests on your solution increment requires the project to adopt a testing strategy that meets the project requirements. The strategy defines the **levels of testing** that will be done and the **test process** to meet acceptance criteria. As a result, the testing strategy defines the environments, infrastructure and test framework needed to support your tests and the way these will be set up and configured.

A data integration project can be tested at different levels. A transformation may be tested individually, or an entire data warehouse (DWH) load may be tested by running it on a known set of source data and evaluating the resulting data. Other tests may be aiming at verifying agreed acceptance criteria, or performance considerations.

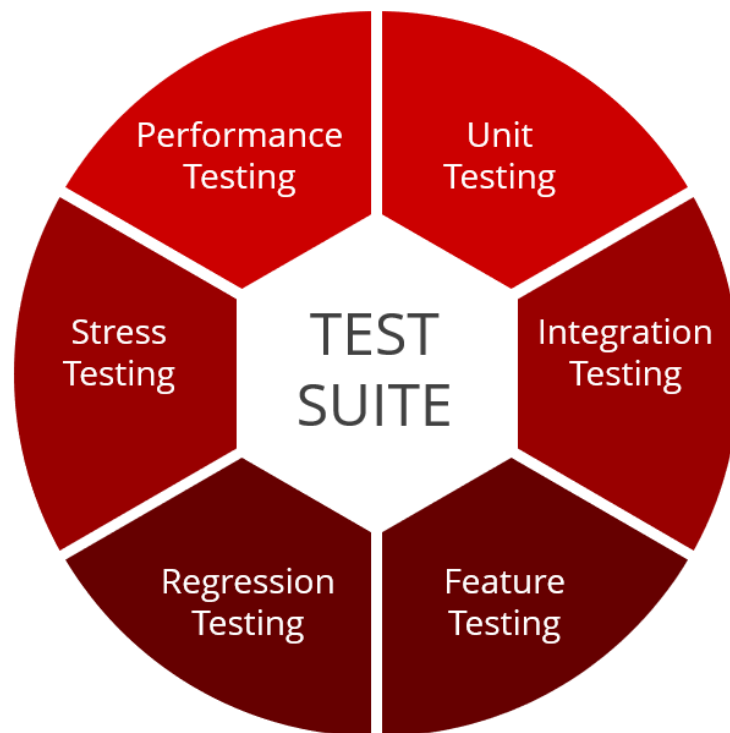


Figure 2: Test Suite

More information is available on these levels in the following sections:

- [Unit Testing](#)
- [Integration Testing](#)
- [Feature Testing](#)
- [Regression Testing](#)
- [Stress Testing](#)
- [Performance Testing](#)



In traditional software development, the most natural way to test code is **unit tests**. In unit tests, small pieces of code, like classes or functions, are tested in small scenarios.

In data integration projects whose logic is mostly embedded in ETL processes, unit tests are usually not very effective. Testing individual ETL jobs and transformations is maintenance-heavy because input data sets need to be prepared, and each calculation and end result of an ETL job must be verified. Since ETL tries to accomplish a lot of processing in a compressed space, testing for everything a piece of ETL does is cumbersome and error-prone, and the effort needed to do it properly may very well outweigh the benefits.

Unit testing does make sense on utility ETL, performing specific isolated tasks and reusable subtransformations.

However, for most of ETL, **logic integration** or **feature testing** is a better choice.

In DI projects, unit testing involves testing a single isolated unit of ETL that performs a computation. In PDI a **unit** is a transformation or subtransformation.

A unit test is a test that is relatively small in size, has a narrow scope, and is easy to write and execute. In general, for unit testing we recommend that you:

- Have small data sets to process in your unit tests and use known input data that is reset for each test. This prevents issues of tests failing for the wrong reasons.
- Have known configurations, meaning that data sources/targets and paths should be set through variables or parameters.
- Have reproducible results.
- Not have any dependencies on outside systems.

An example of a unit test involving utility ETL could be the following:

Imagine our retail business involves calculating customer loyalty points with every purchase. The calculation of these points depends on multiple variables like amount of goods purchased, time of the year, and promotions in effect. Instead of hardcoding this in multiple places of your DI project, this is implemented through a subtransformation. To make sure this unit of ETL keeps functioning correctly after changes happen to the ETL, a unit test will be part of our test suite. This unit test will reset a known input data set, calculate the loyalty points and compare the results against an expected output data set.



Integration testing aims to verify that a logical subset of jobs and transformations works together as intended. Any complex logical step in an ETL process like loading data into staging tables, building a derived business entity in a DWH, or building a star schema, can be the subject of an integration test.

An integration test usually:

- Brings the input data into place
- Runs the appropriate subset of jobs and transformations
- Verifies aspects of the outcome, which may include:

- The ETL must succeed.
- It must produce certain output data.
- It may have to produce some log files with expected content.

This is the most appropriate approach for testing most ETL logic in data integration projects. Its main challenge is that a consistent realistic set of source data must be maintained to make the ETL process operate in a way that resembles production operation as closely as possible.

In reality, your data integration project will consist of multiple main jobs, key areas of your solution. For example, your DWH project might consist of a staging and a warehouse job. Each of those jobs could be the subject of an integration test.

Integration tests should also be written around expected behavior during failure conditions. For example, How does the ETL behave if:

- Files are missing or corrupted
- Databases are down or contain unexpected data
- An ETL job is run with unexpected parameters

Integration tests are written to verify sensible behavior, which is solution-specific (retries, reconnects, log warnings, halts, and so on).

The integration tests do a more convincing job of demonstrating the system works than unit testing would. However, in most cases integration tests lie within the realm of the developer. The customer is not able to launch this kind of test for reasons such as lack of deep knowledge of the solution.




Often, a data integration project will have a set of features or properties listed as part of the project's requirements list. Tests that cover these are usually called **feature tests**, but are integration tests by their nature. They are useful in the acceptance phase of the project.

These are often the tests the customer can most directly relate to, because they are phrased in terms agreed on during scope and requirement management discussions. Feature tests don't concern themselves with intermediate results or side effects, just the result.



Sometimes a bug is discovered that is worth keeping an eye on. Writing a test that specifically guards against a known bug reappearing is called **regression testing**.

Regular integration and feature testing are usually a good guard against these bugs. However, if the bug is difficult to test regularly or depends on conditions that are hard to reproduce in regular tests, a regression test is a good way of making sure that the bug can't reappear unnoticed.



Stress Testing

Under which conditions does the solution break? This is an interesting aspect of every data integration project. Every solution breaks eventually, under the right circumstances. **Stress tests** are designed to find out what the limits are, and make sure the solution does not regress beyond acceptable limits.

Stress tests establish unusual preconditions or unexpected situations before running parts of the solution, and then verify the solution behaves reasonably. Stress tests are usually not scheduled to run as part of the regular test suite, but are instead often scheduled to run once overnight, since they tend to take more time to complete.

Examples for possible stress tests:

1. What happens if the main ETL job is executed concurrently? The test may try to start five of them as quickly as possible and verify that four of them fail gracefully immediately. That procedure would be repeated a couple of times. Any spurious failures in this test indicate some sort of race condition that needs to be taken care of.
2. What is the minimum amount of memory Pentaho Data Integration needs to complete a typical run? Start the solution with a conservative minimum like 1GB and see if, and when, it starts stalling. Any jumps in memory requirements will be identified and can be examined if this test is done.
3. If an ETL transformation is reading BLOB fields from a database, how big do they have to get for PDI to go out of memory? Start the test with an average length of your binary field and see if and when it starts stalling when you increase the length.



Performance Testing

Performance is often an important aspect of data integration projects. It is worth running a realistic workload for the solution on a regular basis and recording the time it took to complete (database logging takes care of that). If the performance starts to deteriorate, it is often easier to find the underlying cause by looking at the commits around that same time.

Performance tests are usually also not scheduled to run as part of the regular test suite. They are often scheduled to run once overnight instead, together with the stress tests.

Continuous Integration Infrastructure

Apart from deciding on the right testing strategy, CI also requires the right infrastructure to be available to support you in continuously integrating and testing code changes made by the development team. Among other aspects, your CI infrastructure consists of a:

- [Test environment](#)
- [CI server](#)
- [Test framework and suite](#)

Test Environment

The test environment is an isolated environment where the CI server runs the test suite. This environment must contain all the pieces of infrastructure that are required by the process being tested. It is a *shared-nothing* space, where the ETL solution must be able to run end-to-end, without causing disruption for anybody if it fails. That means database instances are in place, all data sources like SFTP servers are there, and any working directories for local storage are available.

Preparing a test environment is usually a non-trivial task, as test databases, test files, and the Pentaho software itself would need to be part of it, together with the CI server. Even more complex infrastructure, like a Hadoop cluster, is often required. Therefore, it is important to stick to an automated solution for building and deploying a test environment.

If the test environment needs to change, that change is documented in version control. If any experimental changes or misbehaving tests cause the test environment to become corrupted, the test environment is easily restored. At the same time, the automated scripts building the test environment provide a complete and unambiguous list of requirements and prerequisites for the project to run and successfully pass the test suite, which is very valuable when moving to production.

CI Server

Automated testing requires a CI server, which is the main component in the CI pipeline, orchestrating and organizing the entire process from a commit to a tested solution package ready to be used or distributed.

Jenkins is CI software that helps automate software development and is one of the most common tools used for CI within different Pentaho deployments. Here are a few of its main features:

- Server-based system that supports many VCS integrations (such as SVN and Git)
- Generates output in testing, such as solution package building
- Open source and can be adapted as needed
- Facilitates the integration of the components present in a CI architecture

Test Framework and Suite

The test environment will also host the **test framework**, which prescribes the default way tests are structured and executed and how results are compared to the expected outcome.

In addition, the test environment will also host the **test suite**, which is the complete set of automated test cases for your DI solution. Beyond test cases, a good test suite consists of a set of shared data fixtures (data inputs) reused by multiple tests.

Steps for Test Structure

In data integration projects, a good test is structured around the following steps:

1. Reset the solution database.
2. Load the data fixture appropriate for the test.
3. Run the portion of the solution to be tested.
4. Verify the expected state by many small assertions, each verifying a single aspect of the expected outcome.

Any code that is repeatedly needed for tests should be factored into easy-to-use helpers, part of your test framework. For example, there should be helpers to reset the database, load a fixture and run ETL, so the test remains easy to read, and is not obscured by low level calls to database client tools or shells.

Elements of Test Framework

The test framework comprises a combination of practices and tools that are designed to help developers and QA professionals test more efficiently. The framework should provide a set of guidelines, rules and tools to:

- Guarantee solution development happens according to standards.
- Support the team to write test cases in a consistent way.
- Support test-related activities like loading external resources, and to allow for reuse of other commonly used helper functionality.
- Allow running parts of the test suite selectively: individual tests, or a group of tests.

In general, you need a test framework to help make your test automation solution more reusable, maintainable and stable.

Tests should be easy to write for all developers on the team, so simple scripting languages or domain specific languages are typically preferable to frameworks in more sophisticated languages. It is important that at least one team member be intimately familiar with the test framework. Writing and structuring tests efficiently can be a complex task, and it is important to follow best practices right from the start. Java is always available on Pentaho project environments, so any test framework in a language that runs on the JVM is an option.

Using PDI to Test PDI

As an **alternative approach** for the test suite and framework, it often makes sense to **use PDI to test PDI**. This means that your test suite and framework could contain tests and helpers which are actually written as PDI jobs and transformations testing pieces of your actual DI solution.

If, for example, a subtransformation is expected to modify certain fields in a stream, it is relatively easy to embed it into a test transformation that verifies the output is as expected, and have the transformation fail if it is not. These kinds of test jobs are transformations are best included into the test suite by having the test just run the test job or transformation and verifying PDI's exit code.

Look at it this way: you could create a PDI transformation with a bounded data set as input and execute the procedure (in this case, using a mapping transformation). At the end, the process compares the procedure result with the expected value and decides if it works properly. To load the data fixture and to compare the results, helper jobs or transformations can be included in your test framework:

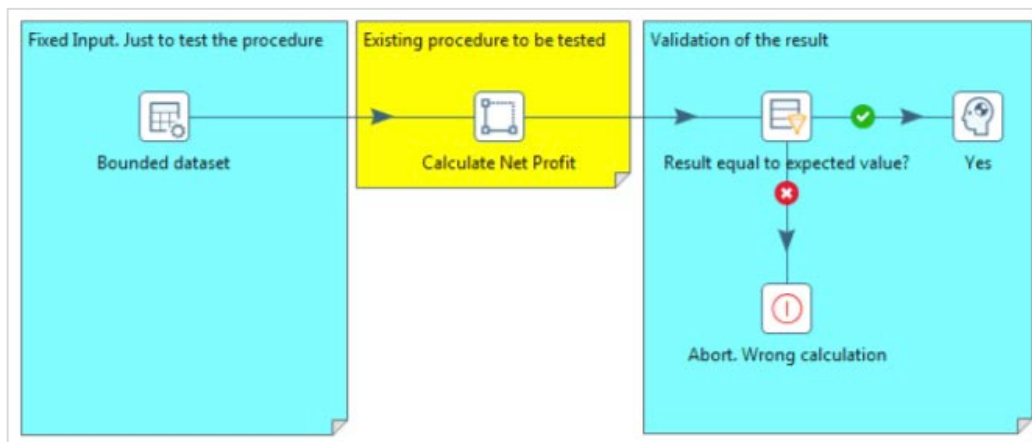


Figure 3: Use PDI to Test PDI Simplified Example

We also recommend that your test framework use standard **output formats for testing results**. Using a reporting format such as JUnit XML output allows you to incorporate those results into Jenkins and establish success/health factors to determine if a build satisfies expectations.

Continuous Integration with Pentaho

The following high-level diagram shows the end-to-end workflow between various stages in the CI process and highlights the positioning of the main tools used in the field.

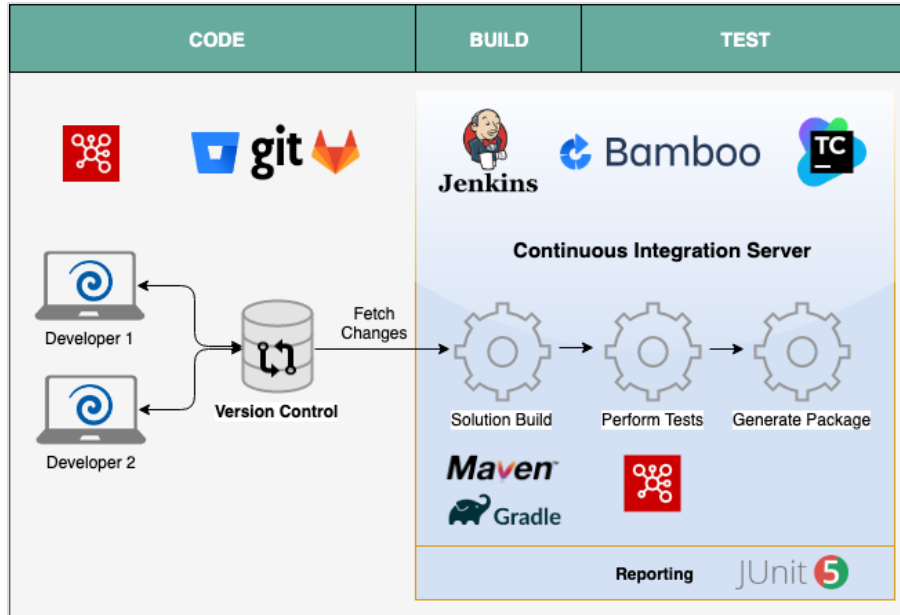


Figure 4: CI Workflow

As a starting point, this workflow recommends the following stages:

1. Multiple developers develop PDI jobs and transformations and commit their solution to a shared solution repository hosted in a VCS like BitBucket, GitHub or GitLab.
2. The CI solution runs in your test environment, hosting among others the CI Server, test framework and suite. A CI Server like Jenkins, Bamboo or TeamCity will automate your build and test pipeline.
3. Solution builds, supported by tools like Maven or Gradle, are optional in our workflow and depend on your specific solution components. There is no need to build the Pentaho jobs or transformations, because they are interpreted at execution time.
4. Next, the CI server executes your test suite and test results could be reported back to the CI Server in a format like JUnit.
5. As a last step, the solution package is generated, ready to be deployed to the integration server for UAT tests.

This workflow and some of the tools to support and automate it are explained in more detail in the [Configuring the CI Server Workflow](#) section of this document. More information is available in the following sections:

- [Project Foundations](#)
- [Test Suite and Framework Definition](#)
- [Configuring the CI Server Workflow](#)
- [Next Steps](#)

Project Foundations

Onboarding on your project's DevOps journey starts with setting up the right project foundations for your PDI solution on which you can build your development, CI and CD strategy. Such project foundations consist of a set of best practices for starting your PDI project, including project structure and the project's technical governance and lifecycle management strategy.

All these best practices regarding setting up your project foundations are discussed in detail in our [PDI Project Setup and Lifecycle Management](#) best practices document. Its main points are:

- **Version Control System:** A team collaborating on a DI solution requires a shared repository of artifacts. 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.
- **Content and Configuration Management:** Your DI solution runs in different environments as it moves through the development and test lifecycle. This means that you need to separate the PDI artifacts and code (for example, the PDI jobs and transformations) from the configurations (the connection details, directory paths and other variables) that make the code run properly
- **Structured Solution Approach:** Starting a successful DI project involves working in a structured and repeatable fashion which is agreed on by the team. This includes agreeing on a structure for your content and configuration repositories and various standards that define how the development and governance of the solution will be managed. Some of those structures and approaches should be captured in a **DI framework**. 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.

Test Suite and Framework Definition

The PDI Project Setup and Lifecycle Management document introduces the project foundations concept details through a sample `sales_dwh` project which is included in the supporting demo, together with a detailed description of the DI framework.

In this section, we extend the `sales_dwh` sample project with actual tests for the project. We also extend the DI framework with a test framework, helping to structure your tests and make them more efficient by introducing helper functionalities. The test framework will also take care of executing your automated tests by integrating with a CI server.

As a starting point, familiarize yourself with the PDI Project Setup and Lifecycle Management document, the concepts described within and the `sales_dwh` project demo and DI framework discussed there. The following diagram summarizes the additions we will make to the DI framework and project setup to include the test suite and framework functionality:

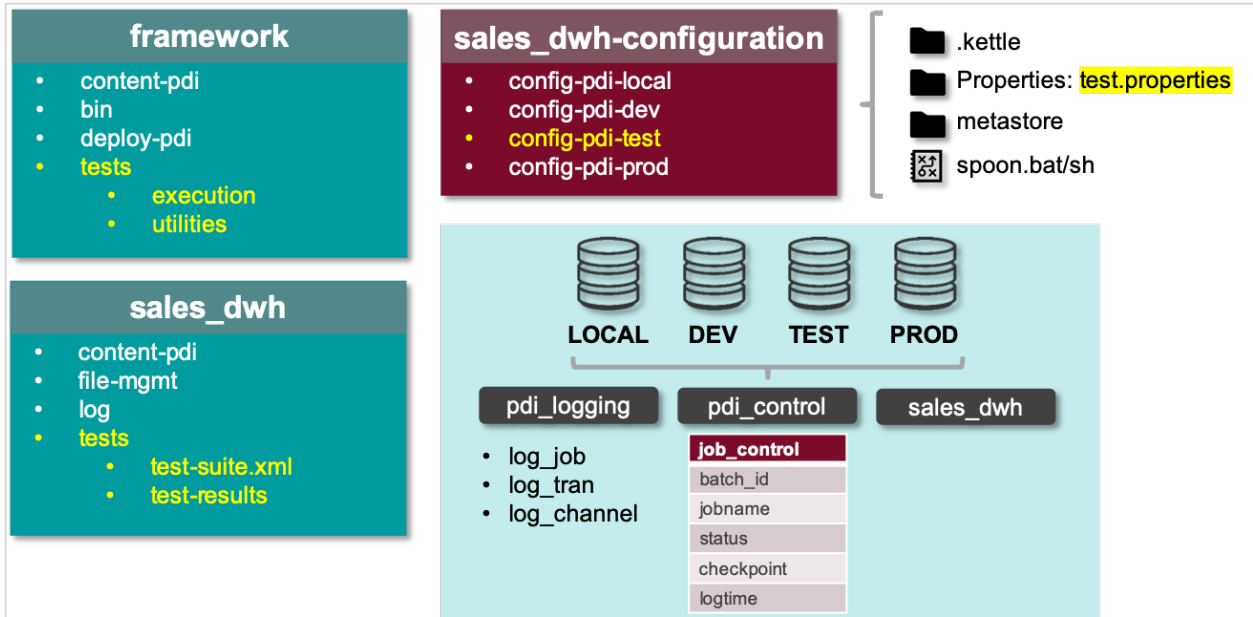


Figure 5: Test Suite and Framework Summary

config-pdi-test Environment

Our automated testing solution will be hosted by the test environment. The configuration details of this environment need to be added to the configuration repository for the `sales_dwh` project.

Under the `properties` folder, we will introduce one more properties file called `test.properties`. This file will be used to introduce test-specific variables for the project.

Tests Folder

The project's test suite, the combination of test jobs and transformations, will be housed in the project's `etl` repository. We will not use the `content-pdi` folder for this, since these tests will stay on the local filesystem rather than being deployed to the Pentaho Repository.

We will create a new folder in the `etl` repository called `tests`. This folder will be the home for three types of artifacts:

1. **Test jobs and transformations:** the developer will name these.
2. **test-suite.xml:** an abstraction between the test jobs and transformations in this folder and the test framework later on, this file indicates which tests need to get executed for this project and in which order.

The file contains the following attributes per test:

Attribute	Details
Name	The title of the test case
Order	The order of execution
Location	The location of the test job or transformation, relative to the <code>tests</code> folder
Type	<code>kjb</code> or <code>ktr</code>
Class	Allows you to create grouping within the test results later in the document. Each dot introduces a new group.
Package	This will always be <code>test</code> .

An example XML file is:

```

<test-set>
  <test name="jb_environment_test"
    order="1"
    location="jb_environment_test"
    type="kjb"
    class="environment.validations"
    package="test"
  />
  <test name="tr_webservice_test"
    order="2"
    location="tr_webservice_test"
    type="ktr"
    class="environment.validations"
    package="test"
  />
  <test name="jb_main_sales_dwh_staging_test"
    order="3"
    location="jb_main_sales_dwh_staging_test"
    type="kjb"
    class="solution.validations"
    package="test"
  />
</test-set>

```

- test-reports folder:** whenever a new test is executed, the test result XML file will be stored here

To make this `tests` folder's location known to the test framework, we introduce a new variable in the `test.properties` file called `TEST_HOME`:

File	Variable	Description
<code>test.properties</code>	<code>TEST_HOME</code>	<code>\${ROOT_DIR}/sales_dwh/tests</code> This always points to a location on the filesystem, since only the <code>content-pdi</code> part of the projects will move to the Pentaho Repository.

Test Case Example

Since we will use PDI to test PDI, your test suite and framework will contain tests and test helper functionality, written as PDI jobs and transformations, that test pieces of your actual DI solution.

The design of those test jobs and transformations will depend on the level of testing that you are doing and on what exactly needs to be confirmed. However, the general idea is always that your job or transformation needs to succeed for a positive test result and fail for a negative test result. Later in this document, we will see how the test framework makes this test result translation to the CI server.

An example test case could be an environment check on whether we can successfully connect to our three project databases: `sales_dwh`, `pdi_logging`, and `pdi_control`. The test job `jb_environment_test.kjb` is located within the project's `TEST_HOME` location and performs two things: loading environment variables and checking the DB connections. The job will fail if the test is not successful.

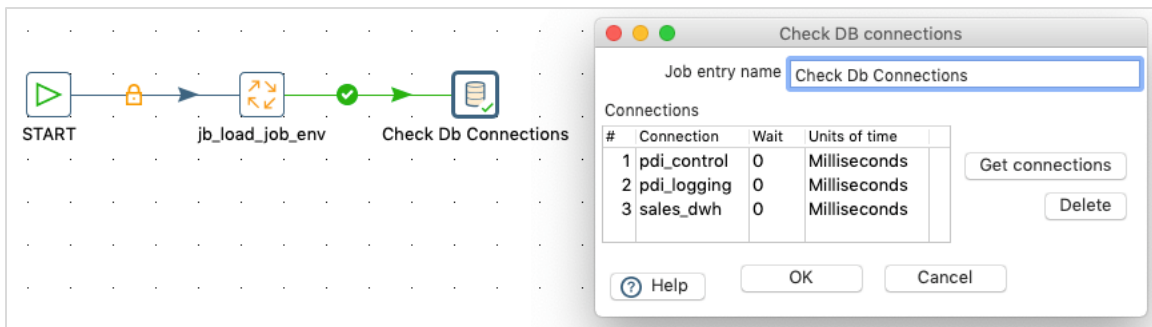


Figure 6: `jb_environment_test.kjb`

Other test cases might need a test data set as input and will have outputs that need to be compared with an expected outcome. To load such data fixtures and compare the results, helper jobs or transformations can be included in your test framework.

Test Framework

We will use Jenkins as the CI server to automate the execution of our test cases and to build a test pipeline. However, we will need the test framework as a wrapper around our individual test cases to facilitate the integration with our CI server.

The test framework will be part of the DI framework folder, located in the `tests` folder (`framework/tests`). The folder has two subfolders:

- **execution:** main jobs and transformations taking care of the test suite execution
- **utilities:** helper jobs and transformations that individual test cases can use to make their tests more efficient

This overview shows the jobs and transformations in the test framework:

```

|-- framework
|   |-- tests
|       |-- execution
|           |-- jb_main_test_executor.kjb
|           |-- tr_test_executor.ktr
|           |-- tr_exectute_test_job.ktr
|           |-- tr_exectute_test_tran.ktr
|       |-- utilities
    
```

1. The test execution process starts with the `jb_main_test_executor` job located in the execution folder. The job has a single input parameter:

Parameter	Description
<code>P_PROJECT_NAME</code>	Name of the project for which you want to execute the test suite

2. The job starts with the `tr_log_batch_id` transformation from the DI framework, loads the project's `test.properties` file, and launches the `tr_test_executor` transformation (`${FRAMEWORK_DIR}/tests/execution/tr_test_executor.ktr`):



Figure 7: `jb_main_test_executor.kjb`

3. Next, the `tr_test_executor` transformation continues with loading the project's test suite, executing the test cases in the specified order and building the JUnit test results XML file based containing the test case results (`${TEST_HOME}/test-reports/results_final.xml`):

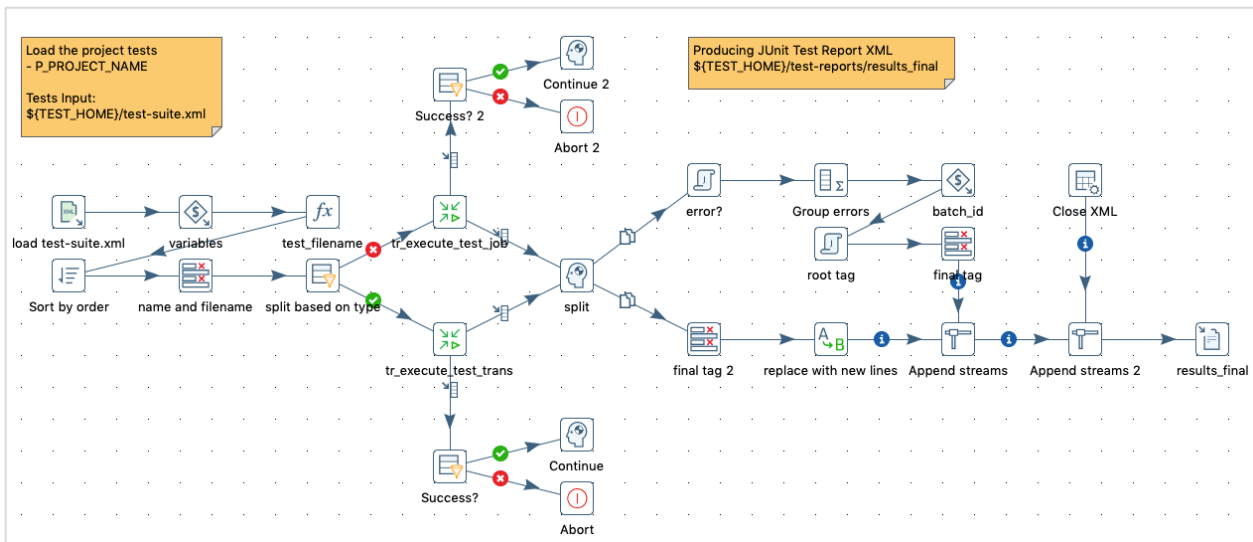


Figure 8: `tr_test_executor.ktr`

- To execute the test cases, the transformation uses the `tr_execute_test_job` or `tr_execute_test_trans` transformations, depending on whether the test case is a job or a transformation. Both do this in a very similar way, so we will only look at the `tr_execute_test_job` transformation in detail:

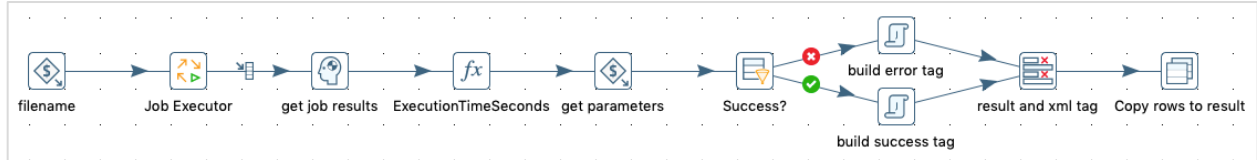


Figure 9: `tr_execute_test_job.ktr`

- After the `tr_execute_test_job` transformation executes the actual test case job, it continues processing the execution results to construct the error or success XML tag containing execution result details.

Example success tag:

```

<testcase name="jb_environment_test" classname="environment.validations"
package="test" time="0.148">
<system-out><![CDATA[2019/08/22 12:59:32 - jb_environment_test - Starting
entry [jb_load_job_env]
2019/08/22 12:59:32 - jb_load_job_env - Using run configuration [Pentaho
local]
2019/08/22 12:59:32 - jb_load_job_env - Starting entry [wtl-
load_project_environment]
2019/08/22 12:59:32 - *** INFO: LOADING PROJECT AND JOB ENVIRONMENT *** -
2019/08/22 12:59:32 - jb_load_job_env - Starting entry [Success]
2019/08/22 12:59:32 - jb_load_job_env - Finished job entry [Success]
(result=[true])
2019/08/22 12:59:32 - jb_load_job_env - Finished job entry [wtl-
load_project_environment] (result=[true])
2019/08/22 12:59:32 - jb_environment_test - Starting entry [Check Db
Connections]
2019/08/22 12:59:32 - jb_environment_test - Finished job entry [Check Db
Connections] (result=[true])
2019/08/22 12:59:32 - jb_environment_test - Finished job entry
[jb_load_job_env] (result=[true])
]]>
</system-out>
</testcase>

```

Example error tag:

```
<testcase name="tr_webservice_test" classname="environment.validations"
package="test" time="">0.204
<failure type="type" message="ERROR Found" />
<system-out><![CDATA[2019/08/22 13:35:50 - Generate rows.0 - Finished
processing (I=0, O=0, R=0, W=1, U=0, E=0)
2019/08/22 13:35:50 - Generate random value.0 - Finished processing (I=0,
O=0, R=1, W=1, U=0, E=0)
2019/08/22 13:35:50 - Abort.0 - ERROR (version 8.3.0.0-371, build 8.3.0.0-
371 from 2019-06-11 11.09.08 by buildguy) : Row nr 1 causing abort "Abort
and log as an error": [2], [587505556], [0]
2019/08/22 13:35:50 - Abort.0 - ERROR (version 8.3.0.0-371, build 8.3.0.0-
371 from 2019-06-11 11.09.08 by buildguy) : Not able to connect to
Webservice
2019/08/22 13:35:50 - tr_webservice_test - Transformation detected one or
more steps with errors.
2019/08/22 13:35:50 - tr_webservice_test - Transformation is killing the
other steps!
]]></system-out></testcase>
```

- The `tr_test_executor` transformation is configured to stop processing test cases the moment there is a failure in one of the test cases. This behavior is configurable with the following illustrated steps:

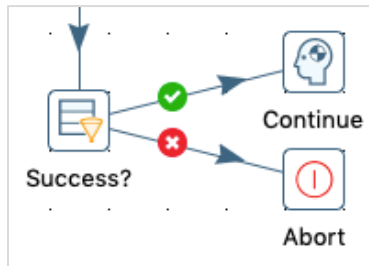


Figure 10: Abort After Failure

- Next, the `tr_test_executor` transformation builds the `results_final` XML file. In order to generate that final XML file, the transformation adds a header and a footer tag. The header tag contains details about the test suite name, which includes the job log batch ID, and the number of tests:

```
<testsuite name="Test Suite Batch ID 55" tests="3" failures="1" skipped="0"
errors="0">
... test case results ...
</testsuite>
```

Configuring the CI Server Workflow

In this section, we will build on the test framework and suite and show how to complete the CI workflow by introducing the Jenkins CI server and how to use and configure it for our `sales_dwh` project.

Demo Starting Point

We recommend you commit the demo repositories to your personal or company Git or alternative VCS account so you can later connect the repositories to Jenkins in order to be able to run the local demo.



If you want to use a VCS other than Git, keep in mind that these instructions may be inappropriate for that alternative VCS. For advice on VCS-specific integrations, contact Hitachi Vantara.

For the next steps of the CI demo description, we assume the following locations for the demo repositories:

- [Framework](#)
- [Sales Data Warehouse](#)
- [Sales Data Warehouse Configuration](#)

Installing Jenkins

Once you have set up the starting point for the demo CI solution:

1. [Install Jenkins](#) on the same machine.
2. Make sure to install the following Jenkins **plugins**. Not all plugins can be installed during initial installation, so add the rest once Jenkins is up and running:
 - a. GitHub
 - b. JUnit
 - c. File Operations
 - d. Performance
3. Next, make the necessary **configuration changes**, if needed, including the port number.
4. Find out where the default **Jenkins workspace** is located.

The workspace directory is where Jenkins “builds” your project. It contains the repositories Jenkins checks out from the VCS, plus any files generated by the build itself. Remember that with Pentaho jobs and transformations, we are not actually building anything. We will see later that a **build** is simply an execution of the project that you define in Jenkins.

Configuring config-pdi-test

Start from the files provided in the demo and adapt them to your situation. Make sure you commit your changes to Git.

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

Jenkins Continuous Integration Server

The Jenkins CI server is the main component in our CI pipeline, orchestrating and organizing the entire process from a commit to a tested solution package ready to be used or distributed. We will use Jenkins to automate the execution of our test cases, using the test framework, and to build the end-to-end test pipeline.


The following sections will describe the necessary steps to setup the CI workflow for the `sales_dwh` project. The first step is to create a CI project for Jenkins. The following sections break this down into distinct parts that you can better understand each stage.

Creating a Jenkins Project

Once Jenkins is up and running, start by creating a Jenkins project (freestyle project). It is important to name your Jenkins project the same as your DI project name: `sales_dwh` in our case.

Enter an item name

» Required field



Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Figure 11: Creating a Jenkins Project

Source Code System Definition

The Jenkins server needs to be able to access the source code, that is, our project artifacts. Jenkins offers capabilities to integrate with many VCSs like Git and Subversion through native plugins, allowing you to connect directly to a specific URL and branch. This makes Jenkins the workspace of your project once the checkout/clone task is done.

To configure the source code:

1. Select your `sales_dwh` CI project and click **Configure**.
2. Find the **Source Code Management** tab and select the radio button for your option.
3. Enter the Repository URL, credentials, and other information to set up integration.

Please note that we are selecting the `master` branch for the demo solution. However, for your project, this will be the branch selected applicable to your automated tests.

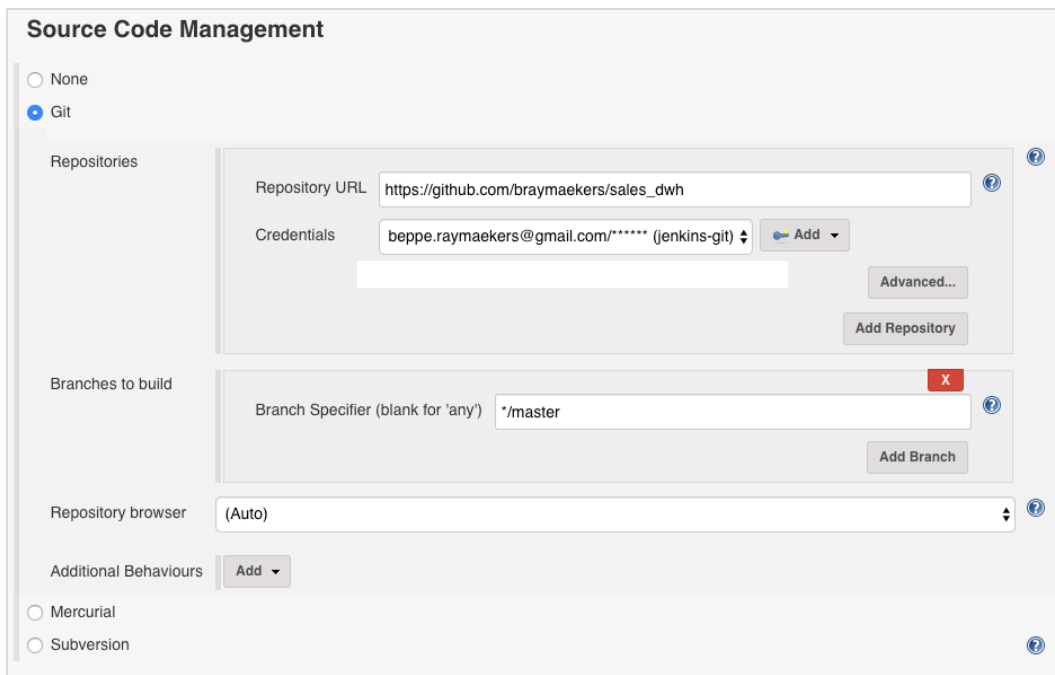


Figure 12: Jenkins Source Code Management

Once Jenkins is connected to the VCS, it will perform a checkout of this repository to the workspace with every build. Per build, Jenkins also reports the Git changes that were committed.

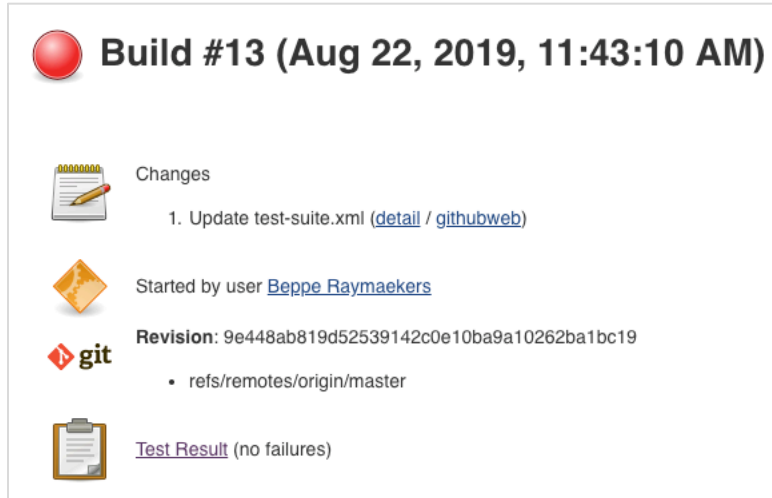


Figure 13: Git Changes Reported by Jenkins Per Build

However, our CI project is dependent on more than one Git repository. In order to also have the `sales_dwh-configuration` and `framework` repositories available within the Jenkins workspace, we need to **initialize** those **remaining repositories** there.

1. Change to your Jenkins workspace location.
2. Use the following commands to initialize the repositories:

```
git clone https://github.com/braymaekers/sales_dwh-configuration
git clone https://github.com/braymaekers/framework
```

Trigger the Process

A project execution (called a `build`) retrieves the latest version of the project artifacts from the centralized code repository, but there are different options on how you can trigger the process, based on your needs and CI maturity:

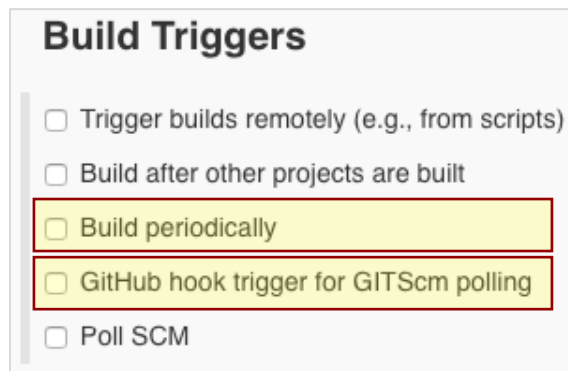


Figure 14: Jenkins Project Build Triggers

- **Build periodically:** You can schedule the process to be executed based on a time rule that follows the syntax of `cron` to build periodically. For example, you can set the rule to `nightly` if you want nightly builds.
- **GitHub hook trigger:** The CI server tools offer the ability to actively listen for repository changes through a plugin, initiating a build when the tool detects changes. Jenkins has such a plugin for GitHub.

For the `sales_dwh` CI project, we will not set any triggers for now, and will run our project manually.

Building the Code and Performing Tests

You do not need to actually compile the Pentaho jobs and transformations, because they are being interpreted at execution time.

However, if you need to resolve dependencies, define targets, or organize the logical distribution of the solution, we recommend using a product like Apache Maven.

For our DI project, the main premise is that PDI is used to test PDI. The development of the solution must include the creation of jobs and transformations to test the behavior, take a sample of input data, use developed functionality, and evaluate if the results match as expected.

Make sure you run environmental tests such as these, before running ETL tests:

- Can I connect to the database?
- Is the Hadoop cluster up and available?
- Can I run a MapReduce process?

The main ETL artifacts to test are those that are used in many parts of the solution:

- Mapping transformation (subtransformation)
- Metadata injection-driven execution
- Job and transformation execution

There are many ways to **execute the automated tests within Jenkins**. Although you can use plugins like Maven Surefire to perform automatic testing capabilities, we actually prefer using PDI's Kitchen to execute the test framework and suite. However, in general we recommend you use the tools you require depending on the level of maturity and the volume and type of tests needed.

In order to run our test framework that will execute our project's test suite, we will use the Kitchen client available in the test environment (`config-pdi-test`). This will execute the PDI client on the same machine where Jenkins also resides. We will execute the test-framework and test suite from the local filesystem.

To configure the build:

1. Select your `sales_dwh` CI project and click **Configure**.
2. Find the **Build** tab, click the **Add build step** button and select the **Execute shell** option.

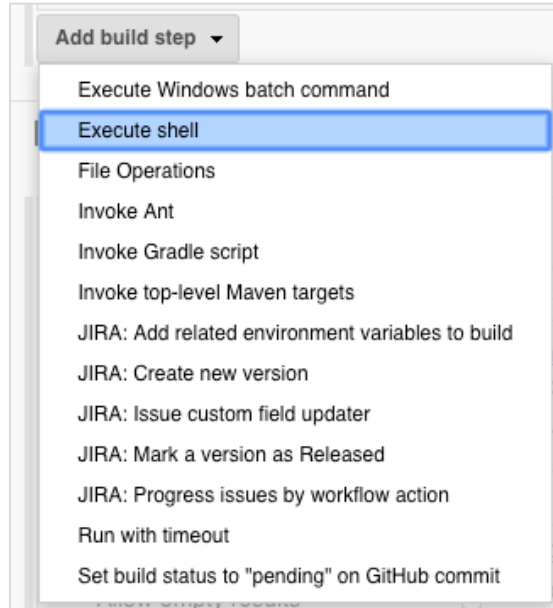


Figure 15: Add Build Step

3. Enter the following command in the **Command** textbox:

```
cd ${WORKSPACE}/../sales_dwh-configuration
git pull

cd ${WORKSPACE}/../framework
git pull

cd ${WORKSPACE}/../sales_dwh-configuration/config-pdi-test
sh kitchen.sh "-
file=${WORKSPACE}/../framework/tests/execution/jb_main_test_executor.kjb"
"-param:P_PROJECT_NAME=sales_dwh"
```

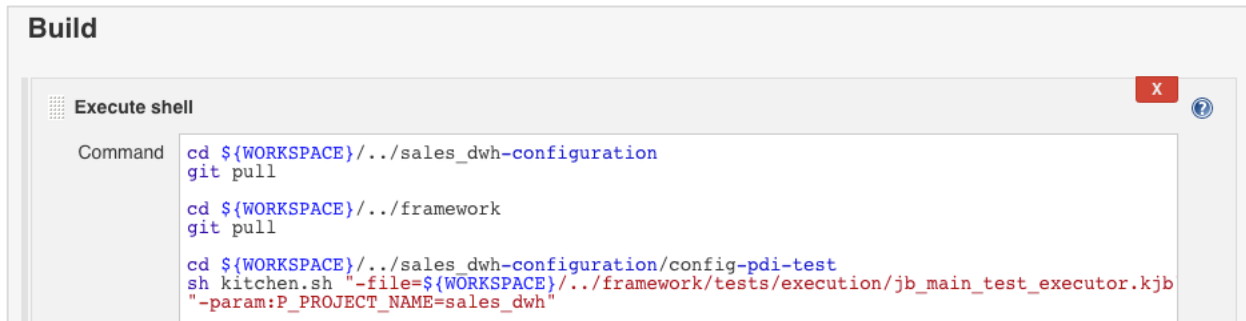


Figure 16: Execute Shell Build Step

Let's have a closer look at the shell command. Before we start running the test framework through kitchen, we first need to get the latest status of the `sales_dwh-configuration` and `framework` Git repositories.

Remember that we already connected the `sales_dwh` Git repository to the Jenkins project using the Git plugin. As a result, this repository will always be synchronized to the Jenkins workspace whenever you trigger a build of the project. However, the Git plugin only allows you to sync a single repository.

This means that the other repositories that we need in the workspace are not synced automatically, so we do this manually in the command.

After that, we use the `kitchen` shell from the `config-pdi-test` environment configuration folder to start the `jb_main_test_executor` job from the test framework. This job requires a single parameter, `P_PROJECT_NAME`, which is set to `sales_dwh`.

Evaluate Testing Results

We recommend using standard output formats for testing results. Using a reporting format as a [JUnit XML output format](#) allows you to incorporate those results into Jenkins and establish success/health factors to determine if the test suite satisfies test expectations.

Remember that the `jb_main_test_executor` job is already producing the JUnit test summary file containing the detailed test results per test case. We only need to integrate it with the Jenkins project, using the post-build actions of our project.

To configure the post-build action:

1. Select your `sales_dwh` CI project and click **Configure**.
2. Find the **Post-Build Actions** tab, click the **add post-build action** button and select the **Publish JUnit test result report** option.

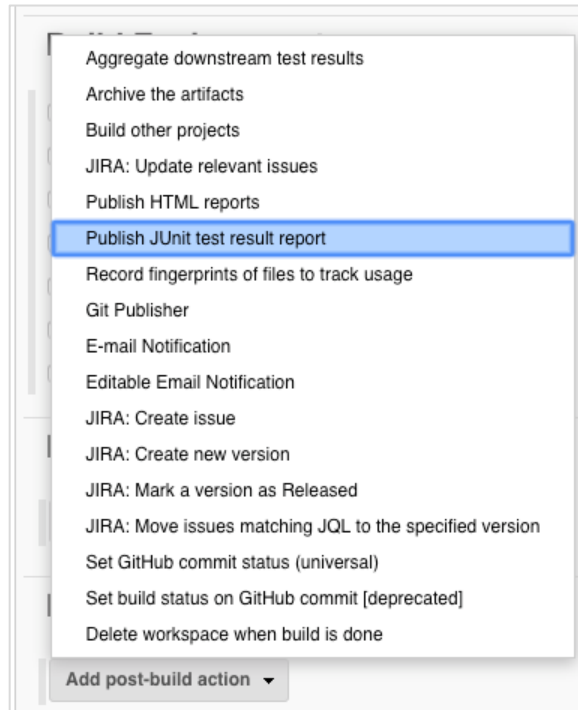


Figure 17: Add Post-Build Action

3. Configure the action as follows:

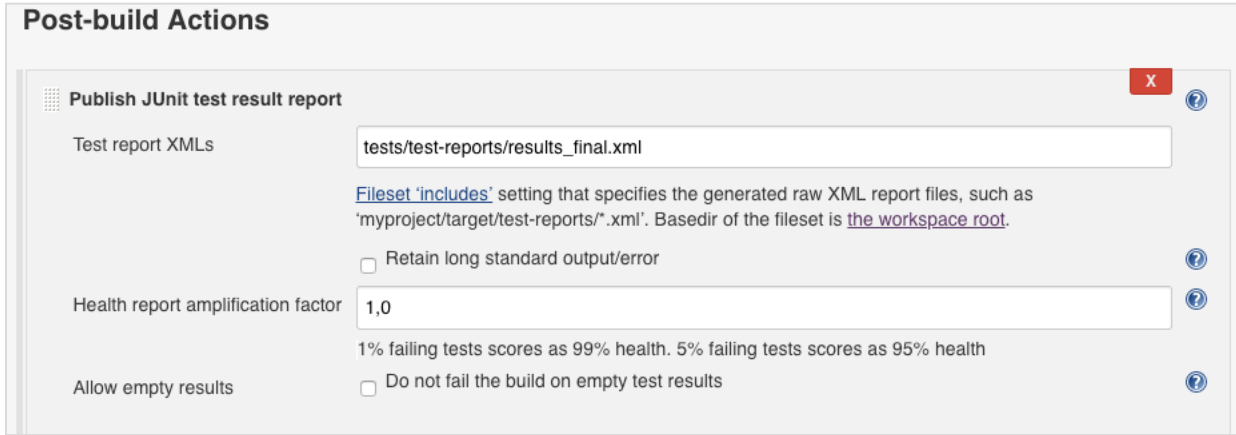


Figure 18: Publish JUnit Test Result Report

What we are actually doing is telling Jenkins where to find the JUnit XML file, which is relative to our project’s location within the workspace.

This `results_final.xml` will notify Jenkins of the **execution status of our test cases** within the test-suite. Next to that, Jenkins will retrieve the **execution status of the build step** of the project using the [exit codes](#) associated with the Kitchen command line.

In the Jenkins portal, you can see this in the **Build History** view on your left when you open the `sales_dwh` CI project:

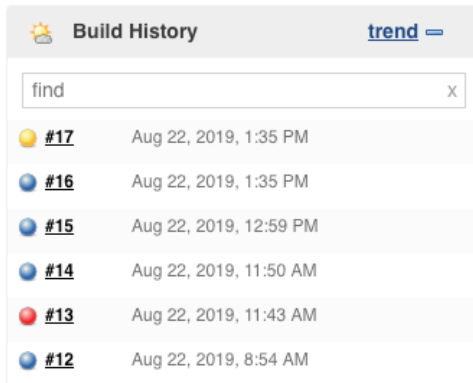


Figure 19: Build History

The colors on the left indicate the following:

- a. **Blue:** The `jb_main_test_executor` job executed successfully, and all test cases finished successfully.
 - b. **Red:** The `jb_main_test_executor` job failed, and something is wrong in your solution configuration.
 - c. **Yellow:** The `jb_main_test_executor` job executed successfully, but some test cases failed.
4. Once you **drill-through to the details of a build**, you will see the following screen:

Figure 20: Build Details

5. To see the **details of the execution**, you have two options:
 - a. Click on the **Console Output** on the left to via the Kitchen logging of the job execution:

```

2019/08/22 08:35:17 - Append streams 2.0 - Finished processing (I=0, O=0, R=5, W=5, U=0, E=0)
2019/08/22 08:35:17 - results_final.0 - Finished processing (I=0, O=5, R=5, W=5, U=0, E=0)
2019/08/22 08:35:17 - jb_main_test_executor - Finished job entry [tr_test_executor] (result=[true])
2019/08/22 08:35:17 - jb_main_test_executor - Finished job entry [sv-test.properties] (result=[true])
2019/08/22 08:35:17 - jb_main_test_executor - Finished job entry [tr_log_batch_id] (result=[true])
2019/08/22 08:35:17 - jb_main_test_executor - Job execution finished
2019/08/22 08:35:17 - Kitchen - Finished!
2019/08/22 08:35:17 - Kitchen - Start=2019/08/22 08:35:08.619, Stop=2019/08/22 08:35:17.482
2019/08/22 08:35:17 - Kitchen - Processing ended after 8 seconds.
    
```

Figure 21: Console Output

- b. Click on **Test Result** on the left or middle to see the test result details. In this view, you will see the grouping that you created within the `test-suite.xml` file with the `class` XML element value. Every `dot` creates a new level:

Package	Duration	Fail (diff)	Skip (diff)	Pass (diff)	Total (diff)
environment	0,13 sec	1 +1	0	1 -1	2
solution	2,2 sec	0	0	1	1

Figure 22: Jenkins Test Result Overview

- You can continue drilling down to see the test case details. Note that you can see **the Job batch ID** within the test suite name:

Failed

environment.validations.tr_webservice_test (from Test Suite Batch ID 63)

Failing for the past 1 build (Since 🟡 #11)
[Took 0 ms.](#)
[add description](#)

Error Message

ERROR Found

Standard Output

```

2019/08/22 08:35:14 - Generate rows.0 - Finished processing (I=0, O=0, R=0, W=1, U=0, E=0)
2019/08/22 08:35:14 - Generate random value.0 - Finished processing (I=0, O=0, R=1, W=1, U=0, E=0)
2019/08/22 08:35:14 - Abort.0 - ERROR (version 8.3.0.0-371, build 8.3.0.0-371 from 2019-06-11 11.09.08 by buildguy) : Row nr 1 causing abort "Abort and log as an error": [2], [2032626386], [0]
2019/08/22 08:35:14 - Abort.0 - ERROR (version 8.3.0.0-371, build 8.3.0.0-371 from 2019-06-11 11.09.08 by buildguy) : Not able to connect to Webservice
2019/08/22 08:35:14 - remainder.0 - Finished processing (I=0, O=0, R=1, W=1, U=0, E=0)
2019/08/22 08:35:14 - Abort.0 - Finished processing (I=0, O=0, R=1, W=1, U=0, E=1)
2019/08/22 08:35:14 - tr_webservice_test - Transformation detected one or more steps with errors.
2019/08/22 08:35:14 - tr_webservice_test - Transformation is killing the other steps!
2019/08/22 08:35:14 - even int.0 - Finished processing (I=0, O=0, R=1, W=1, U=0, E=0)
                
```

Figure 23: Test Case Details

Solution Package Generation

Your resulting output should be a ready-to-use package with the solution after building and testing is finished. The package must be ready to be incorporated in the automatic deployment pipeline or to be deployed manually as part of your company criteria.

This packaging process can be done through script and plugins within Jenkins, or you can incorporate Apache Maven capabilities as part of the build lifecycle. For the `sales_dwh` CI project, we will use the **File Operations Jenkins Plugin**.

To configure the solution packaging:

- Select your `sales_dwh` CI project and click **Configure**.
- Find the **Build** tab, click the **Add build step** button and select the **File Operations** option.
- Click the **Add** button and select **File Zip** and **File Rename** operations.
- Configure them as follows. Note that you can leave the folder path for the File Zip operation empty, since that will default to zipping the project folder within the workspace.

Figure 24: File Operations Build Step

- For the File Operations build step configuration we make use of [Jenkins variables](#). The ones we are using are:

Variable	Description
WORKSPACE	The absolute path of the workspace, this includes the project folder
JOB_NAME	Name of the Jenkins project
BUILD_NUMBER	The current build number

- For the demo solution we move the package zip files into a folder called `build_artifacts` in the Jenkins workspace folder:

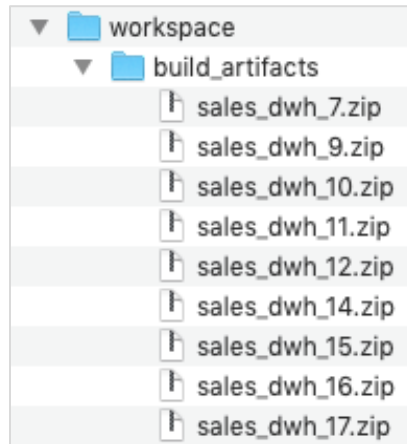


Figure 25: Build Artifacts folder

CI Server Workflow Summary

Starting from the test framework and suite, we continued with integrating those with the Jenkins CI Server where we set up a project which implements our CI workflow for the `sales_dwh` project:

- We integrated the project with the `sales_dwh` Git repository so we could add triggers to the build process: whenever changes happen to the repository, scheduled or manual.
- We configured the build part of the workflow where we are synchronizing the other Git repositories (`sales_dwh-configuration` and `framework`) and using the kitchen client from the `config-pdi-test` environment to execute the test framework and suite for our project
- We configured Jenkins to know where to find the resulting JUnit test results file so we can visually see testing progress within Jenkins.
- We finally packaged the solution for later deployment purposes into the UAT environment.

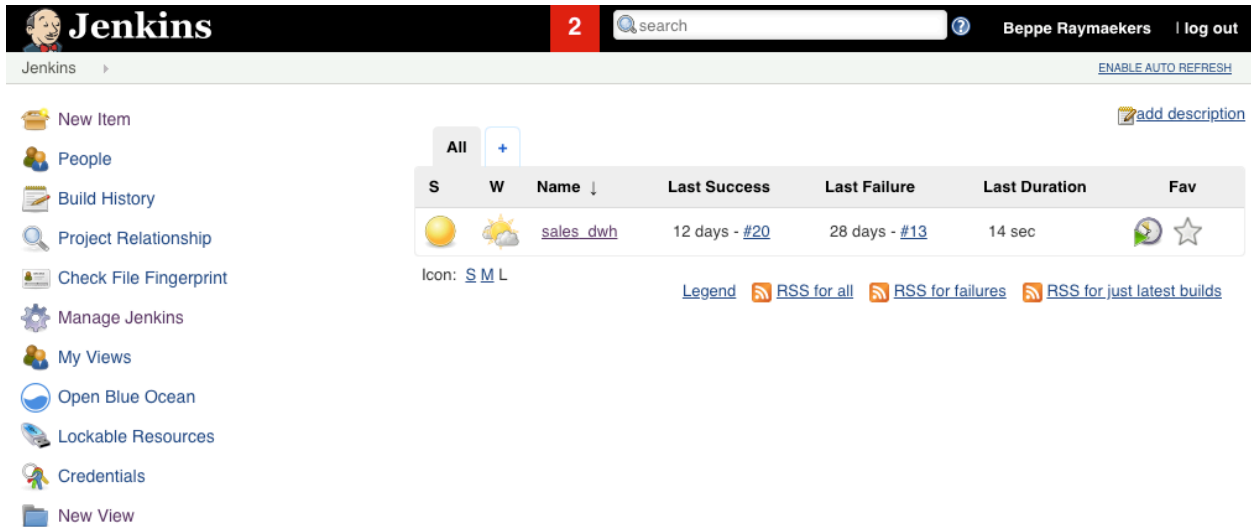


Figure 26: Jenkins Dashboard

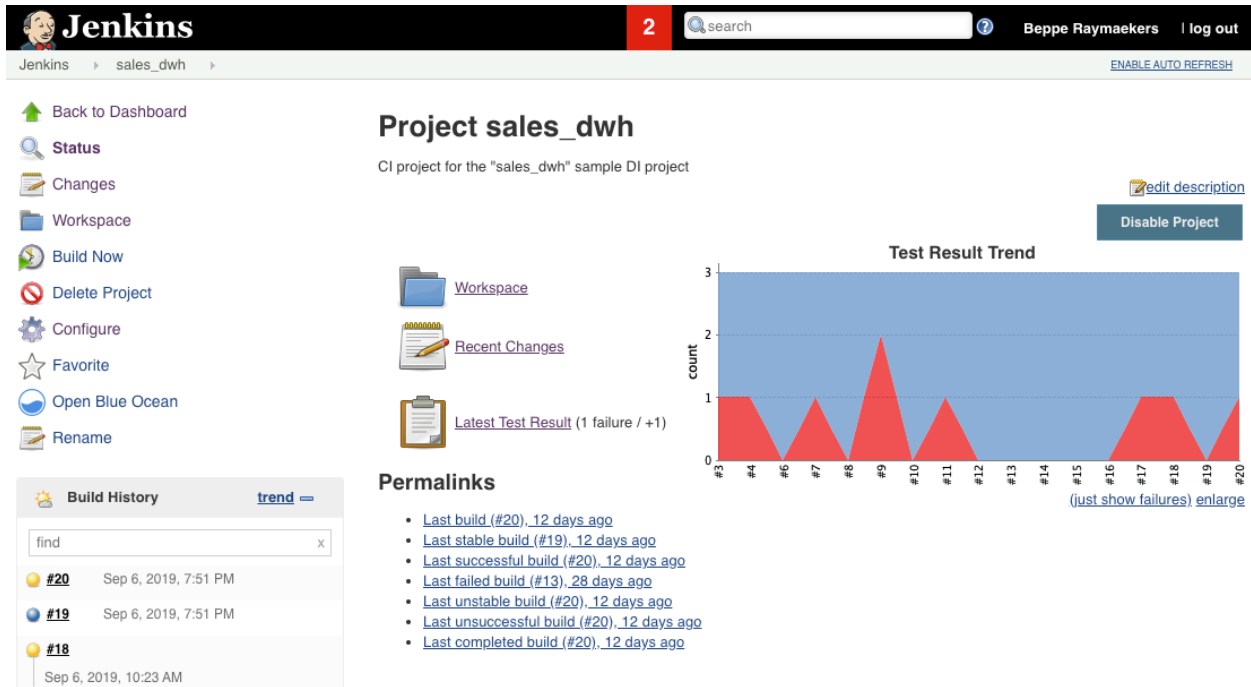


Figure 27: Jenkins Project Overview

Possible Extensions to CI Workflow Setup

This document covered some best practices on CI with PDI, focusing on the building blocks to get you started. Depending on your specific project requirements, you can customize or extend this starting point. Possible extensions of the current CI workflow include:

- Make use of a dedicated testing framework like Apache Maven Surefire.
- Integrate your test results into a project tracking application like Jira.
- [Xray Test Management for Jira](#) is a powerful plugin for Jira to visualize and manage your test cases and test executions. It will be straightforward to integrate this with the current setup since Xray can interpret the JUnit XML file to see which tests got executed and what their execution status was.

The screenshot displays the Jira Test Management interface for a test plan titled "Test Plan for version 1.0". The interface is divided into several sections:

- Test Plan Details:** Includes fields for Type (Test Plan), Priority (Medium), Status (Unresolved), and Assignee (Bruno Conde).
- Overall Execution Status:** A progress bar showing 3 PASSED, 2 FAILED, and 1 TO DO.
- Test Steps (3):** A list of test steps with their respective actions, data, and expected results.

Step	Action	Data	Expected Result	Step State
1	Attachments @ (1) I choose the operation of the calculator Subtraction		The operation must appear selected.	PASSED
2	I enter the input into the calculator	I1: 5 I2: 2		PASSED
3	I press the Calculate button		The result 3 should be displayed in the screen, on the right of the "=" sign.	FAILED
- Activity:** A section at the bottom showing the execution history, including a tooltip for Step 3: "CALC-15 Error while subtracting 2 numbers".

Next Steps

So far, we have been focusing on the Continuous Integration maturity phase within the wider DevOps lifecycle. Although this is a crucial starting point, it does not end here. We stopped at generating the solution package, but in order to automate the complete workflow from developing the solution, to testing it, all the way to a successfully deploying the solution into production, more steps are needed.

More documents will be introduced in the PDI DevOps Series to share best practices on Continuous Delivery and Continuous Deployment, including alternatives to previously discussed methods.

Related Information

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

- [Demo Files](#)
- [Installing Jenkins](#)
- [Jenkins: Building a software project](#)
- [JUnit Format](#)
- [Kitchen Status Codes](#)
- [Pentaho Components Reference](#)
- [Pentaho Data Integration Best Practices Library](#)
- [Pentaho Installation](#)
- [Testing strategies for data integration](#)
- [Xray Test Management for Jira](#)