Top Ten Best Practices in Oracle Data Integrator Projects

By FX on Jun 25, 2009

Top Ten Best Practices in Oracle Data Integrator Projects

This post assumes that you have some level of familiarity with Oracle Data Integrator. Please refer to the ODI Tutorial for a quick introduction, or to the complete ODI documentation for detailed information.

Oracle Data Integrator (ODI) is a very powerful product when handled correctly. Unfortunately, some mistakes may lead to dramatic results in Integration projects. This post compiles the Top 10 best practices that avoid the most common mistakes seen in integration projects made with Oracle Data Integrator.

Best Practice #1 - Understand and Use Correctly Topology and Contexts

The ODI topology and the contexts are one of the most powerful feature for running your design-time or run-time artifacts in various environments.

In ODI, all developments as well as executions are performed on top of a *Logical Architecture* (Logical schemas, logical agent), that resolves, in a given *Context* to a *Physical Architecture* (real/physical source/targets data servers/schemas and ODI run-time agents). Contexts allow you to switch the execution of the artifacts from one environment (context) to another.

Now, read the previous paragraph again. Make sure you got the concept of context, logical and physical architecture.

Two common mistakes are made with it comes to contexts:

- Forgetting to perform the logical/physical architecture mapping in a given context. This is a topology administration mistake that leads to executions not working in a given context. To avoid this, just make sure that all the logical resources are mapped to physical resources in this context.

- A big one is *forcing the context when not needed*. In interfaces or procedures, there are context selection boxes, set to *default* or *execution context*. Well, unless you are really willing to force the context for a functional reason, leave these selection boxes as they are. There are few use cases where the context needs to be forced.

In a Nutshell: Make sure you understand the Topology and Context concepts perfectly. Take care of topology definition and context mapping. Avoid forcing contexts.

Best Practice #2 - Context-Independent Design

This is my top #2 mistake. In many ODI artifacts (procedure, variables, interfaces, etc..), you can add expressions and SQL code. A very common mistake is to enter qualified object names, as in the following example that drops a TEMP_001 table in a staging schema:

```
DROP TABLE STAGING.TEMP_001
```

Well, this is context-dependent code. If you run this code in the production environment where the staging schema is called PRD_STG, you code does not work. Note that the schema names are defined in the topology, and the contexts would handle getting the right schema name depending on the execution context. The question is: how to use this in your code?

The Substitution Methods (aka OdiRef API) exist to leverage in your code the metadata stored in ODI and make the code context-independent. In this case they will assure that the qualified table name is generating according to the context you are running your code into.

Using Substitution Methods, the generic code would be like:

```
DROP TABLE <%odiRef.getObjectName("L", "TEMP_001","W")%>.
```

Refer to the Substitution Methods Reference Guide for helping you using this API. The expression editor also helps a lot.

In a Nutshell: As soon as you start typing a schema name, a database name, a user name or any information that is data server or schema dependent in the code; Stop; breathe deeply and consider the use of a substitution method.

Best Practice #3 - Use Procedures Only When Needed

Procedures allow you to perform very complex actions, including SQL statements. In addition, they allow you to use source and target connections, and support data binding. In short, you can move data using procedures.

Developers who feel at ease with SQL may be tempted to code their transformations and data movements within procedures instead of using interfaces

There are several issues with using procedures in such way

- Procedures contain manual code that needs to be **maintained manually**.

- Procedures do not maintain cross references to other ODI artifacts such as datastores, models, etc, making their maintenance very complex compared to interfaces.

Procedures should never be used to move and transform data; these operations should be performed using interfaces.

In a Nutshell: When you start moving/transforming data in procedures between data structures, it is likely that you are misusing procedures instead of interfaces. Use interfaces instead.

Best Practice #4 - Enforce Data Quality

Data Integration project leader sometimes do not take into account the quality of their data. This is a common mistake, as data integration itself may end up being moving and transforming garbage, and propagating it over the applications.

ODI allows you to enforce data quality of source data using *static checks* as well as the quality of the data before it is pushed into a target via *flow checks*. Using both these checks, you can make sure that the quality of the data is improved or enforced when the data is moved and transformed.

In a Nutshell: Enforce data quality using both static and flow checks. Data quality is not an option.

Best Practice #5 - Handle Error Cases into Packages

In a package, you can sequence any number of steps. Every single step in a package may fail for some reasons (target or source database is down being one of them, too many errors detected in one interface being another).

You should always consider these different error cases when designing packages.

In a Nutshell: The "OK" path (green arrows) in the packages is a must have, and the "KO" path (red arrow) is what makes your package bulletproof.

Best Practice #6 - Choosing the Right KM

The KM Choice is critical when designing interfaces. KM Choice conditions the features available and the performances of an interface.

Some common mistakes are make in the KM Choice:

- Starting with complex KMs: Beginners want to get their interfaces running quickly, but sometimes do not take into account all the requirements needed for a KM. Choosing for example technology specific LKMs using loaders may lead to an interface not working because of the loader configuration or installation not being correct. A safe choice for starting to work in to use generic KMs (usually SQL KMs) that work in most cases. After designing your first interfaces with these KMs, you can start switching to specific KMs (read their descriptions first!) and leverage all the features of these KMs.

- Over-engineering Interfaces: KMs with extra features have an extra cost in terms of performance. For example, performing simple insert is faster than performing an incremental update. If you are deleting the data in the target before integration, using incremental update is over-engineering, and causes some performance loss. Use the KM that fits your needs.

- Similarly, activating or deactivating some of the KM features may add extra-steps that may degrade the performance. Default KM options are sufficient for running the KM provided out of the box. After running the KM with default options, it is always good to review the options and see if some of them can be changed for your needs. The KM and Option description are a good documentation for understanding the optimizing KM usage.

In a Nutshell: Start with simple KMs, do not over-engineer by selecting complex KMs or activating complex options, and pay attention to the KM options.

Best Practice #7 - Ramp Up on KMs

Knowledge Modules (KMs) is a very powerful framework used at any single point of integration flows in ODI. A large number of KMs are available out of the box and support a large number of database features. Even if it is not needed in most cases, some projects have use cases or requirements that call for KM customization.

So when should you cross the border and start customizing the KMs? The answer is: As soon as you see an operation that need to be performed in each interface (for example, launching a command on the target for optimizing execution).

It is not recommended to start from a blank page when you need a KM. The recommended path is to find the KM that is close to your use case and customize it to your needs.

In a Nutshell: When a specific operation need to take place in many interfaces, do not be afraid to customize the existing KMs and create your own KMs based on the ones provided out of the box.

Best Practice #8 - Set Up Project Organization Early

Project Management may not seem critical when it comes to data integration. Well, it is.

ODI provides many tools that help organizing the development work and project lifecycle: security profiles, development projects, folders, markers, versioning, import/export, printing PDF documentation, etc.

Please review and use all these tools and features for managing the project according to your guidelines. Define the guideline and organization of the project, the naming conventions and everything that will avoid chaos and enforce best practices. Do it from the beginning of the project.

In a Nutshell: Development productivity being high in ODI, it is better to have a strong organization based on ODI feature to avoid a rapidly growing development chaos.

Best Practice #9 - Keep Control on Repository Spawning

In ODI, a master repository can have many work repositories. You can also have multiple master repositories, each of them having his own set of work repositories. Each repository has an ID set at creation time.

Well, a repository is not a document. It is the "source of truth", the central reference that cross-references all the artifacts it contains for easing the maintenance.

In addition, all object being identified by an Internal ID that depends on the Repository ID. These internal IDs uniquely identify an object and are used by the ODI import system. Two repositories with the same ID possibly contain objects with the same internal ID, which mean the *same object* for ODI. Transferring objects between these repositories is similar to copying files with same names across different directories, and may lead to object replacement.

You should make sure that all repositories are created with strictly different IDs (even on different sites), and you should define and document the process for moving objects between repositories using import export, or versioning.

In a Nutshell: Multiplying repositories should be made under strict control and planning (particularly for the repository ID choice), and managing the object transfers using import-export or versioning should be handled via formal procedures.

Best Practice #10 - Be Cautious With the Repository Contents

ODI stores all its information into a metadata repository, stored into a relational database. Once you know this, it is very tempting to start hacking your way through the repository tables to "go faster".

The repository does not implement all the logic that exists in the graphical interface, and does not implement all the business logic that exists in the Java code. Performing request to build for example dashboards or reports on top of the repository is acceptable, but writing or altering repository information is dangerous, and should be left to troubleshooting or support operations, under control of the Oracle Support.

In a Nutshell: Would you do this on your ERP backend database? No! Do not do it with the ODI Metadata Repository.