Building connectors

This page contains tutorials and tips and tricks for building your own MetaModel connectors, e.g. objects that implement DataContext.

First implementation - extend QueryPostProcessDataContext

First we will reuse the abstract class QueryPostProcessDataContext, which makes our implementation a whole lot simpler than starting from scratch.

1. Create a new `XXXDataContext` class that extends `QueryPostProcessDataContext` that holds a reference to a native API object. Taking existing examples:

- `CouchDbDataContext` holds Ektorp' `CouchDbInstance`.
- `HBaseDataContext` holds `HTablePool`.
- CsvDataContext holds a handle to a file (through MetaModel's `Resource` class, to be precise)

2. Implement `materializeMainSchemaTable` to fetch the data that is going to represent the table, using the native API. Good existing examples:

- `CouchDbDataContext` fetches a view will all docs through Ektorp's `CouchDbInstance`.
- `HBaseDataContext` creates a `Scan` of a table through `HBaseTablePool`.
- `CsvDataContext` reads the whole file using our file handle.

3. Return a new `XXXDataSet` instance with the native result passed as a parameter. `XXXDataSet` class takes the native result object and translates it to the MetaModel's `Row` objects.

- `CouchDbDataSet` will parse the JSON document and instantiate MetaModel's `Row` object with this data
- `HBaseDataSet` extracts the values from the Scan and instantiates MetaModel's `Row` object with the values
- `CsvDataSet` translates a line from a CSV file into MetaModel's `Row` object

This is the minimum that needs to be implemented. While we have the native result translated to MetaModel's `Row` objects, selecting specific columns, filtering and so on we get for free from `QueryPostProcessDataContext` class that we subclassed. Of course, it is not the most performant way of querying, for example doing a full scan to get a single row by its primary key is optimized in many databases. Delegating primary key lookups, count queries, queries with simple WHERE clauses to the native API instead of post-processing it in Java is the next step for a developer of a new MetaModel connector.

Optional next step - Override methods for optimized query execution

Now you have a working DataContext, but there may be some really unfortunate performance penalties associated with it.

You may want to look at overriding methods such as `executeCountQuery(...)`, `executePrimaryKeyLookupQuery(...)` and the other argument-variants of `materializeMainSchemaTable(...)` to improve performance and minimize client-side memory consumption too.

Optional next step - Implement UpdateableDataContext

If you wish to support data updates (inserts, updates, deletions etc.) then you have to implement UpdateableDataContext too.

Optional next step - Create a DataContextFactory

If you want to make your DataContext createble in the most convenient way for external tools then it is usually a good idea to implement our SPI interface, `DataContextFactory`. This will enable your DataContext to be instantiated using properties.

Finally, register your factory by creating a file named `/META-INF/services/org.apache.metamodel.factory.DataContextFactory` in your JAR file, and ensure that it's text contents it the fully qualified class name of your factory.