Database Data Loading for Orkit

Extend Orekit with database support to reap the benifits of modern database technology.

Bob Reynders

1 Requirements

In this section I hope to encapsulate all the requirements related directly to the database proposal[1].

Following the policy 'The library user's data and the format thereof exists outside of Orekit' I adjusted the requirements a bit in comparison to the proposal I wrote for SOCIS. The focus is now laid towards creating a flexible design that allows Orekit users to pull in just the dependencies they need to get their own running database populated with Orekit data, instead of the former focus, expanding Orekit to work with *a* database.

Summarized; this document will describe an implementation that is both easy to use *and* flexible for the Orekit user without dragging in unnecessary dependencies.

2 Design

The design I propose to meet these requirements is very different to the one that I described earlier so please read the following section without prejudice.

2.1 Overall Architecture

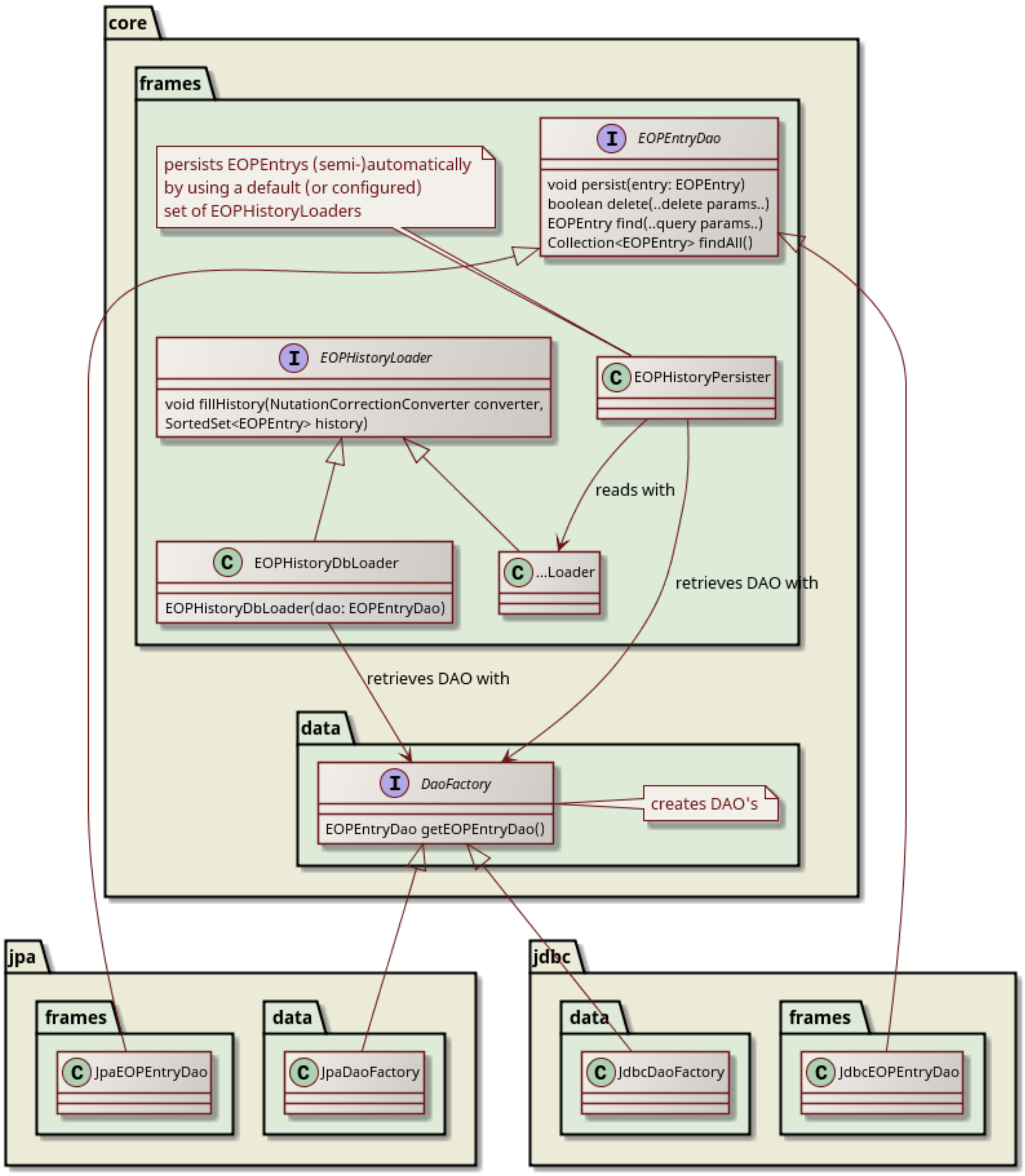
Figure 1 shows the general architecture of database support in Orekit.

Core Design People familiar with the "Data Access Object" (DAO) pattern can recognise its usage in the design.

In order to allow for an interchangeable persistence back-end we remove the coupling from a persistence client (such as an `EOPHistoryPersistenceLoader`) to the persistence implementation. The extra layer in between serves as an abstraction mechanism for transparent changes between persistence implementations.

To do this we create a layer of DAO interfaces, in the EOP example this would be the `EOPEntryDao`. As seen in the design it defines means to persist, delete and

[1]For brevity I do not discuss the additional task of adding a way to reset factory data



core
frames
persists EOPEntrys (semi-)automatically by using a default (or configured) set of EOPHistoryLoaders
EOPEntryDao
void persist(entry: EOPEntry)
boolean delete(..delete params..)
EOPEntry find(..query params..)
Collection<EOPEntry> findAll()
EOPHistoryLoader
void fillHistory(NutationCorrectionConverter converter, SortedSet<EOPEntry> history)
EOPHistoryPersister
reads with
EOPHistoryDbLoader
EOPHistoryDbLoader(dao: EOPEntryDao)
...Loader
retrieves DAO with
retrieves DAO with
data
DaoFactory
EOPEntryDao getEOPEntryDao()
creates DAO's
jpa
frames
JpaEOPEntryDao
data
JpaDaoFactory
jdbc
data
JdbcDaoFactory
frames
JdbcEOPEntryDao

Figure 1: Overall Architecture

retrieve `EOPEntry`s. Concrete implementations of the DAO interfaces are given by actual persistence implementations.

This introduces another problem, the uniform creation of *all* DAO implementations corresponding to one persistence backend. To solve this problem we introduce a DAO factory: `DaoFactory`, it defines methods for all DAOs and its implementation is again specific to the persistence implementation.

With the persistence abstraction in place you probably noticed that the two persistence clients `EOPHistoryPersistenceLoader` and `EOPHistoryPersister` only depend on an implementation of the `DaoFactory` interface. An Orekit user wishing to create an application to (1) persist EOP data and (2) retrieve stored data using the existing `EOPHistoryLoader` scheme is only required to instantiate a `DaoFactory` implementation of his choice.

Modules The outer packages in Figure 1 represent maven modules, respectively, the "core" Orekit module, the "jdbc" persistence module and the "jpa" persistence module. Modules are used to provide an implementation for the persistence interface without flooding the Orekit core with dependencies[2].

2.2 JDBC Module

One of the requirements is to provide an easy to use API that can fit on any custom database setup that might already be in place. Taking this into consideration while designing the JDBC implementation of the persistence interface results in a design shown on Figure 2.

The module provides a corresponding implementation of `DaoFactory` and implements a JDBC specific `EOPEntryDao` class. To allow an existing or custom database scheme, the DAO (and thus the factory) require an instance of an EOPEntryMapping which defines how an Orekit `EOPEntry` is mapped to a database table and its columns.

This module does *not* provide a concrete JDBC implementation but instead requires the user to create the JDBC specific DAO factory with a `DataSource` of its JDBC Driver. Bootstrapping the database support in Orekit "core" with "jdbc" would thus be done by having a JDBC Driver of your choice and instantiating a `JdbcDaoFactory` while using it for the `EOPHistoryPersister` with the `EOPHistoryPersistenceLoader`.

2.3 JPA Module

A JPA implementation would be similar to the persistence part of the earlier proposal with the exception that it will not depend on a JPA implementation but instead on the specification *javax.persistence*.

I have spent today searching for ways to (properly!) allow for Orekit to provide a JPA API that is user configurable so that it can be used with existing databases but all the methods I have found (and there are a few) feel kind of hackish. I do believe that the

[2]The "jdbc" module does not need extra dependencies and is split into the module for the sake of uniformity, see 3.

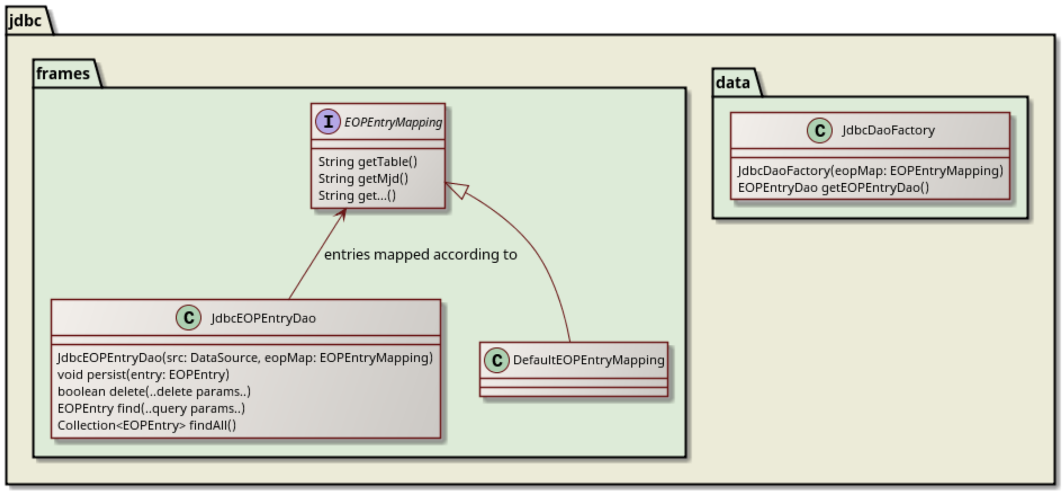

jdbc
frames
data
EOPEntryMapping
String getTable()
String getMjd()
String get...()
entries mapped according to
JdbcEOPEntryDao
JdbcEOPEntryDao(src: DataSource, eopMap: EOPEntryMapping)
void persist(entry: EOPEntry)
boolean delete(..delete params..)
EOPEntry find(..query params..)
Collection<EOPEntry> findAll()
DefaultEOPEntryMapping
JdbcDaoFactory
JdbcDaoFactory(eopMap: EOPEntryMapping)
EOPEntryDao getEOPEntryDao()

Figure 2: JDBC Implementation

JPA persistence module can be useful since it can be used to define an (albeit static) mapping to database tables that can be plugged into existing JPA systems easily.

3 Discussion

Maven Modules To limit dependencies I proposed using maven modules to split Orekit into a multi-module maven project. As maven is not my build tool of choice I cannot expand on the tool support and ease-of-use for this setup, are there any remarks regarding this?

JPA The added functionality of the JPA implementation can be replaced with the functionality of the JDBC implementation if the user resorts to non-specification methods in its JPA setup. Hibernate, EclipseLink and OpenJPA provide means to retrieve its used `DataSource` although its often dirty (e.g. downcasting specification objects to implementation specific objects). Thus, it is less clean but still possible to completely replace the JPA implementation features with just the JDBC implementation features (since its a lower level technology) by lowering ease-of-use. I would like some discussion around this trade-off since I feel its a tricky decision to make.

JDBC Module A JDBC implementation can be provided without requiring additional dependencies since a normal JVM runtime provides *javax.sql.DataSource*. This raises the question if the JDBC implementation should be a separate module or not. Depending on whether or not the JPA module is going to be implemented I feel it should be a separate module to be uniform with persistence implementations. JDBC for example isn't the only persistence implementation that could be written without dependencies, it would be possible to write a Directory+XML persistence implementation or a file-based persistence implementation similar to the way Orekit always works. As with the other topics I would like some opinions on this.

Test Coverage I am a fan of unit testing but under the motto “too much of anything is bad for you” I prefer to check this with you before implementing these. For example, I think it is important to test the JDBC implementation on an in-memory database to see if everything persists and reads well. I would like some opinions on how ’deep’ and how ’strict’ the unit testing should be, should I for example mock `DaoFactory` and test everything isolated from the implementations or not?