Database Data Loading for Orkit

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

Bob Reynders

1 Problem Description

The problem with the current data-loading mechanism is that it is hard to scale (text files) and that a lot of pre-processing is duplicated (parsed information is not stored). A solution would be to stored parsed datasets in a (scalable) database, to do this the existing design should be modified to support database loaders.

Additionally, removing the constraint on data loading for the factory classes consists of exposing a proper API with enough power to (1) remove current data, (2) add new data or (3) reload data.

2 Project Goals

My project goals for this SOCIS task would be:

- Implement an application that uses the existing Orekit features to parse raw datasets and store the results in a database.
- Add support to the existing Orekit feature set to load data from the aforementioned database.
- Extend the current API to support the reloading of data in factories.

3 Implementation

The first stage of implementation will only support persistent EOPEntrys. This document will focus on two main design decisions for implementing persistence, one towards (Open) JPA[1] and the other towards Apache DBUtils[2].

[1]Instead of the EclipseLink implementation I mentioned earlier we can use the Apache implementation for easier licensing http://openjpa.apache.org/

[2]http://commons.apache.org/proper/commons-dbutils/

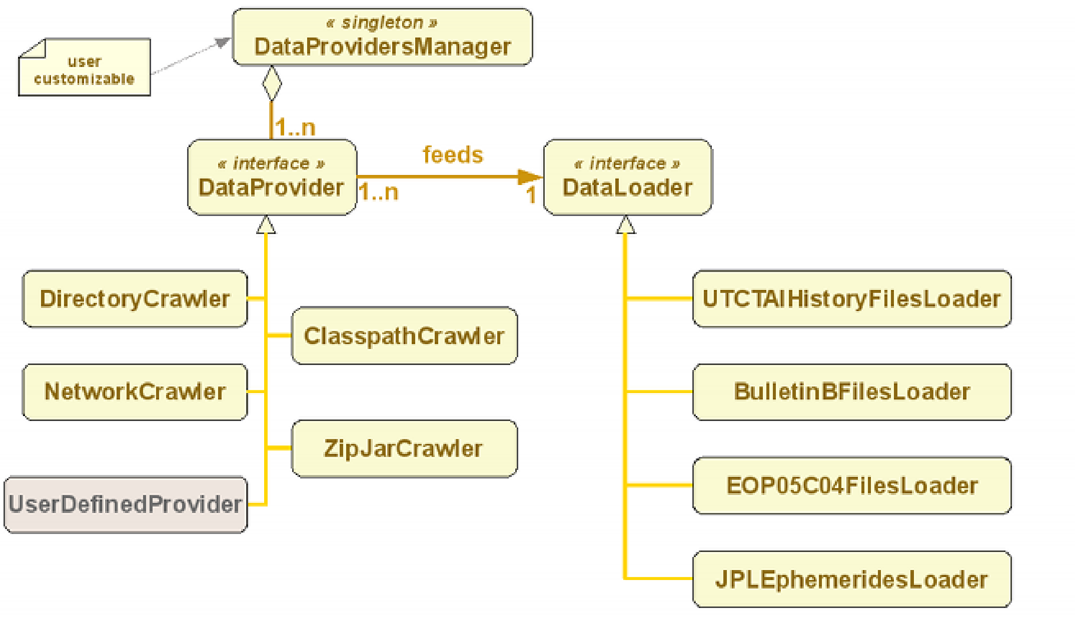


Figure 1: Data handling (https://www.orekit.org/forge/projects/orekit/wiki/Configuration)

3.1 Common Design Description

Both designs are loosely tiered, i.e., there are upper tiers that provide great abstraction and lower tiers that go towards a more fine grained persistence API.

The first tier is already in place for the most part, it contains the existing Orekit classes such as *EOPHistoryLoader* implementations, *EOPEntry*, etc. Additional classes that can be seen as upper tier are the persister classes such as the *EOPHistoryPersister* and the database-dependent implementations of *EOPHistoryLoader*.

The second tier consists of the data access objects, in this case the *EOPHistoryDAO*. Its purpose is to create a layer of abstraction between the actual persistence objects and the access thereof. It will abstract over the lower-level persistence APIs.

The last tier is built up from the persistence classes. In this tier the chosen persistence API is used.

DB Scheme The database scheme that backs the entry storage is shown in Figure 2 and should be self-explanatory.

EOPHistoryPersister Is responsible for (1) loading *EOPEntry*s using several *EOPHistoryLoader* implementations and (2) persisting these entries.

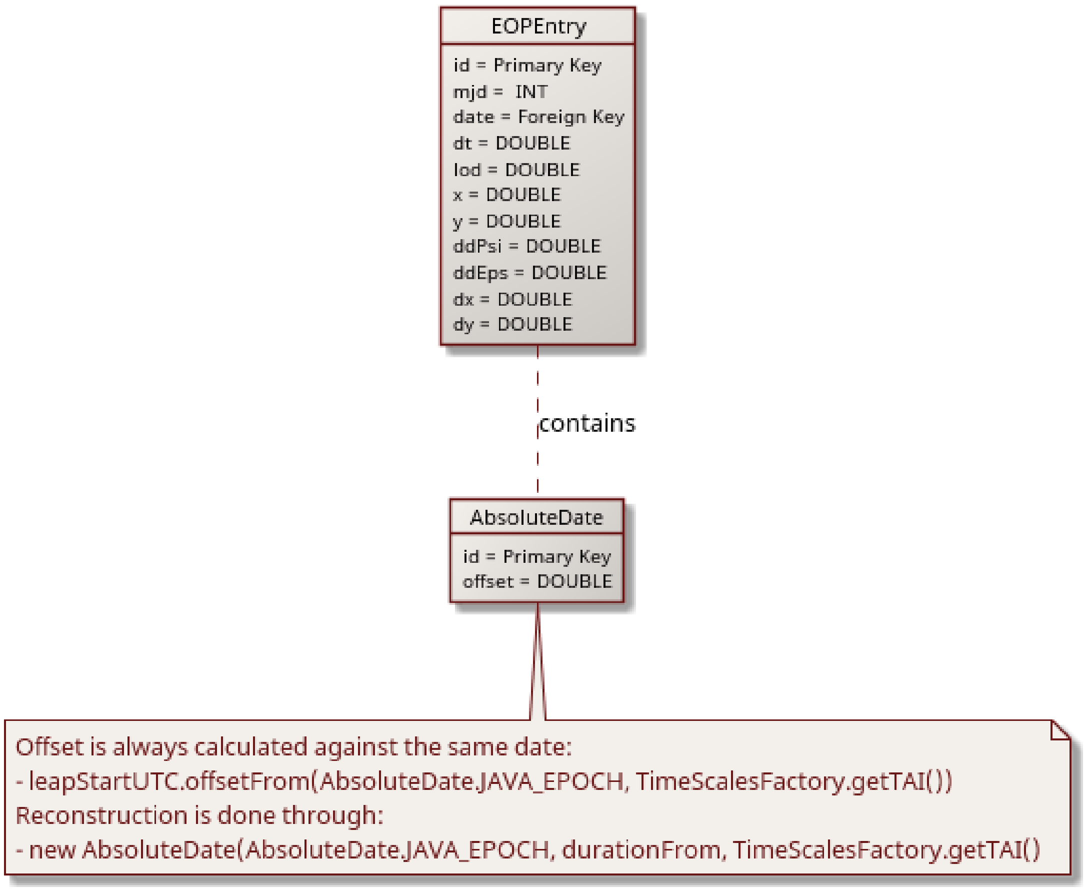

EOPEntry
id = Primary Key
mjd = INT
date = Foreign Key
dt = DOUBLE
lod = DOUBLE
x = DOUBLE
y = DOUBLE
ddPsi = DOUBLE
ddEps = DOUBLE
dx = DOUBLE
dy = DOUBLE
contains
AbsoluteDate
id = Primary Key
offset = DOUBLE
Offset is always calculated against the same date:
- leapStartUTC.offsetFrom(AbsoluteDate.JAVA_EPOCH, TimeScalesFactory.getTAI())
Reconstruction is done through:
- new AbsoluteDate(AbsoluteDate.JAVA_EPOCH, durationFrom, TimeScalesFactory.getTAI()

Figure 2: Database Scheme

Remarks *EOPHistoryLoader* has the fillHistory method defined which takes 2 parameters, a converter and a history set. The converter is used during the actual reading of the *EOPEntry*s and hence will not be used in persistent implementation. I think I might be missing something here, can someone confirm this? If this is the case I suggest moving the converter to the history loader constructor.

3.2 JPA

The JPA model shown in Figure 3 focuses on abstracting away the JPA *EntityManager* usage and implementing specific entity objects that are purely used for persistence.

EOPHistoryDAO Uses *EntityManager*s to persist and retrieve *EOPEntry* instances through the *EntityManagerFactoryManager*.

EOPEntryEntity Is a class that is annotated with the required JPA annotations so that it can be managed and persisted without touching actual SQL.

AbsoluteDateEntity Is another entity with the sole purpose of being part of the *EOPEntryEntity*.

3.3 DBUtils

The DBUtils model shown in Figure 4 uses the DBUtils classes provided to abstract away from the JDBC options.

EOPHistoryDAO Uses *QueryRunner*s and *ResultHandler*s to persist and retrieve *EOPEntry* instances through the *DataSourceManager*.

DataSourceManager Manages the actual datasource.

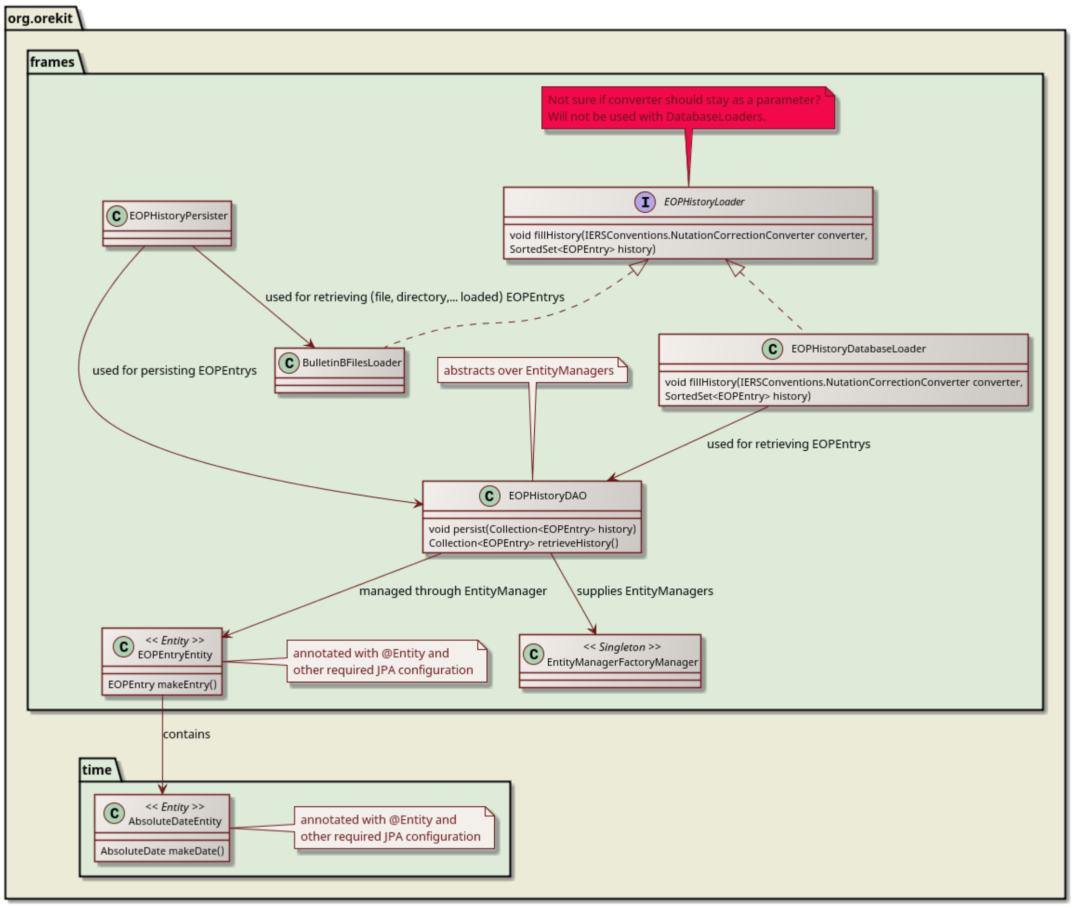
EOPEntryResultHandler Is an implementation that can convert from *ResultList*s to actual *EOPEntry*s.

AbsoluteDateResultHandler Is an implementation that can convert from *ResultList*s to actual *AbsoluteDate*s.

4 Conclusion

I think the main advantage of JPA is that future Java programmers are familiar with it. It is pretty elegant to have your database representation available as Java classes in the form of entities but this arguably takes away from the freedom.

Where DBUtils lacks in ease of use it makes up in fine grained control of SQL queries. If the Orekit team wants to stay closer to SQL I recommend going with this.



org.orekit
frames
Not sure if converter should stay as a parameter?
Will not be used with DatabaseLoaders.
EOPHistoryLoader
void fillHistory(IERSConventions.NutationCorrectionConverter converter,
SortedSet<EOPEntry> history)
EOPHistoryPersister
used for retrieving (file, directory,... loaded) EOPEntrys
BulletinBFilesLoader
used for persisting EOPEntrys
abstracts over EntityManagers
EOPHistoryDatabaseLoader
void fillHistory(IERSConventions.NutationCorrectionConverter converter,
SortedSet<EOPEntry> history)
used for retrieving EOPEntrys
EOPHistoryDAO
void persist(Collection<EOPEntry> history)
Collection<EOPEntry> retrieveHistory()
managed through EntityManager
supplies EntityManagers
<< Entity >>
EOPEntryEntity
EOPEntry makeEntry()
annotated with @Entity and
other required JPA configuration
<< Singleton >>
EntityManagerFactoryManager
contains
time
<< Entity >>
AbsoluteDateEntity
AbsoluteDate makeDate()
annotated with @Entity and
other required JPA configuration

Figure 3: JPA EOP Database Model

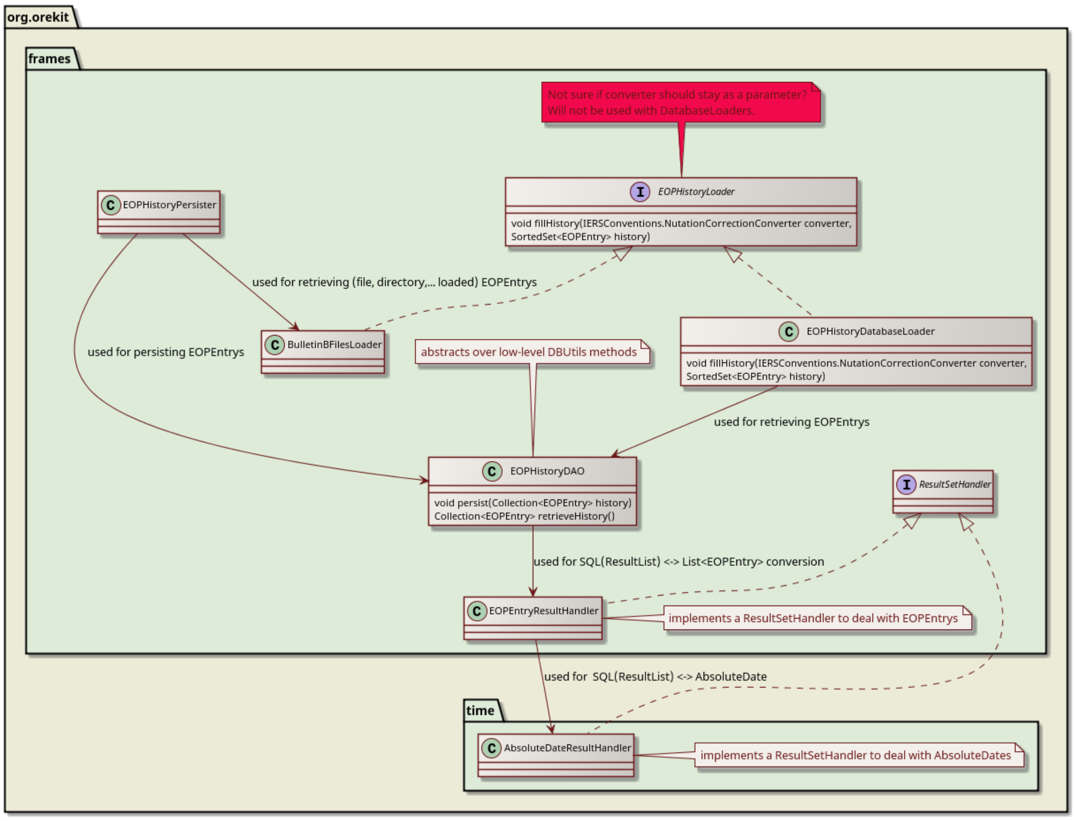

org.orekit
frames
Not sure if converter should stay as a parameter? Will not be used with DatabaseLoaders.
EOPHistoryLoader
void fillHistory(IERSConventions.NutationCorrectionConverter converter, SortedSet<EOPEntry> history)
EOPHistoryPersister
used for retrieving (file, directory,... loaded) EOPEntrys
used for persisting EOPEntrys
BulletinBFilesLoader
abstracts over low-level DBUtils methods
EOPHistoryDatabaseLoader
void fillHistory(IERSConventions.NutationCorrectionConverter converter, SortedSet<EOPEntry> history)
used for retrieving EOPEntrys
EOPHistoryDAO
void persist(Collection<EOPEntry> history)
Collection<EOPEntry> retrieveHistory()
ResultSetHandler
used for SQL(ResultList) <-> List<EOPEntry> conversion
EOPEntryResultHandler
implements a ResultSetHandler to deal with EOPEntrys
used for SQL(ResultList) <-> AbsoluteDate
time
AbsoluteDateResultHandler
implements a ResultSetHandler to deal with AbsoluteDates

Figure 4: DBUtils EOP Database Model

I left out the entire discussion and information I posted earlier about the source of data in the form of inputstream digests. It will be easier to add and explain as an additional feature and I recognise my mistake of tunnel-visioning on it from the start.

Future I think its in our best interest if I can start programming as soon as possible. If I had to pick personally I would go for the JPA implementation because I personally have more experience with JPA. I'm a person who works best when coding and I don't mind altering code to reflect design changes so moving towards DBUtils in the future will not be a problem.