LexEVS 5.x Loader Framework

From Vocab_Wiki

Contents

Introduction

This document is a section of the Loader Guide. It is new in LexEVS v5.1.

Document Purpose

This document provides the detailed design and implementation of the Loader Framework Extension. It is the goal of this document to provide enough information to enable application developers to create custom loaders. This document assumes the developer is already familiar with the LexEVS software.

Loader Framework Background and Enhancements

Previous versions of LexEVS software has provided a set of loaders within an existing legacy framework which served LexEVS developers well over many years. But as LexEVS has gained users, and requests for new loaders have grown, it was decided to create a new loader framework. Specifically, this development work addresses "TASK 6 - IMPROVE LEXEVS LOADING FRAMEWORK" in the National Cancer Institute (NCI) Statement of Work (SOW) document (reference ?????).

The LexEVS v5.1 loader framework meets these emerging needs compared to the loader framework of previous versions:

- is easier to extend
- provides improved performance
- enables dynamic loading of new loaders
- leverages proven open source components, such as Spring Batch and Hibernate

Also, the new framework is completely independent of existing loader code, so there is no impact to existing loaders.

Scope

The LexEVS v5.1 Loader Framework provides a way for LexEVS developers to write new loaders and have them recognized dynamically by the LexEVS code. Also the framework provides help to loader developers in the form of utility classes and interfaces.

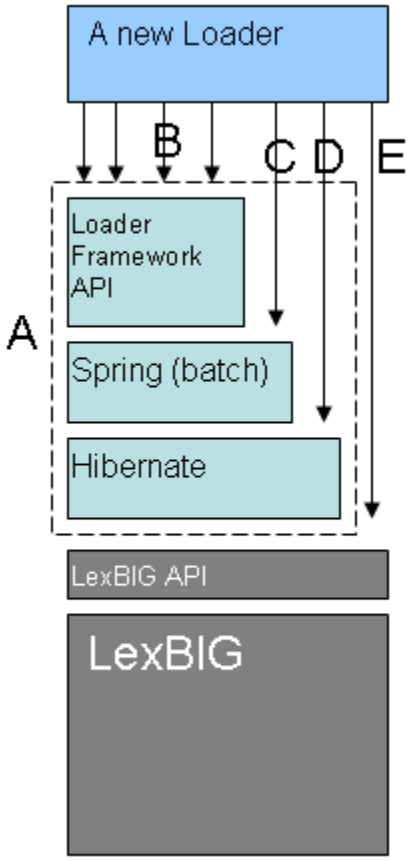
Architecture

The LexEVS v5.1 Loader Framework extends the functionality of LexEVS 5.0. For more information on LexBIG, see LexEVS_Version_5.0.

The image below shows the major components of the Loader Framework.

> (A) A hypothetical new loader in relation to the loader framework, and what expected API usage would be.
> (B) Ideally, the new loader can make most if its API calls through the utilities provided by the Loader Framework API.
> (C) Some work will need to be done with Spring (C) such as configuration of a Spring config file.
> (D and E) It may or may not be necessary for a loader to use Hibernate or the LexBIG API. Again, the hope is that much of the work a new loader may need to do can be accomplished by the Loader Framework API.

The Loader Framework utilizes Spring Batch for managing its Java objects to improve performance and Hibernate provides the mapping to the LexGrid database.



Assumptions

None

Dependencies

- This Loader Framework requires LexEVS release 5.0 or above.
- Development systems are required to install the Sun Java Development Kit (SDK) or Java Runtime Environment (JRE) version 1.5.0_11 or above.
- Maven 2.1 or greater.
- For software and hardware dependencies for the system hosting the LexEVS runtime, refer to Installation and downloads.

Issues

None

Development and Build Environment

Third Party Tools

- Spring: A lightweight open-source application framework.
  - Spring: see [1]
  - Spring Batch: see [2]
  - Sprint Batch Reference: see [3]
- Hibernate: An open source Java persistence framework; see [4]
- Maven: Apache build manager for Java projects; see [5]
- Eclipse: An Open Source IDE. See [6]

Loader Framework Code

The Loader Framework code is available in the NCI Subversion (SVN) repository. It is comprised of three Framework projects. Also at the time of this writing there are three projects in the repository that utilize the Loader Framework.

Loader Framework Projects

- PersistanceLayer: a Hibernate connector to the LexBIG database
- Loader-framework: a framework that sets up build information for Maven
- Loader-framework-core: a framework that contains all the interfaces and utilities; also contains an extendable class "AbstractSpringBatchLoader" that all new Loaders should extend

Loader Proejcts Using the New Framework

- abstract-rrf-loader: a holder for common rrf-based loader code
- meta-loader: a new loader to read the NCI MetaThesaurus
- umls-loader: a loader for reading Unified Medical Language System (UMLS) content

Maven

The above projects utilize Maven for build and dependency management. Obtain the Maven plugin for Eclipse at [7]

How to Use the Loader Framework: A Roadmap

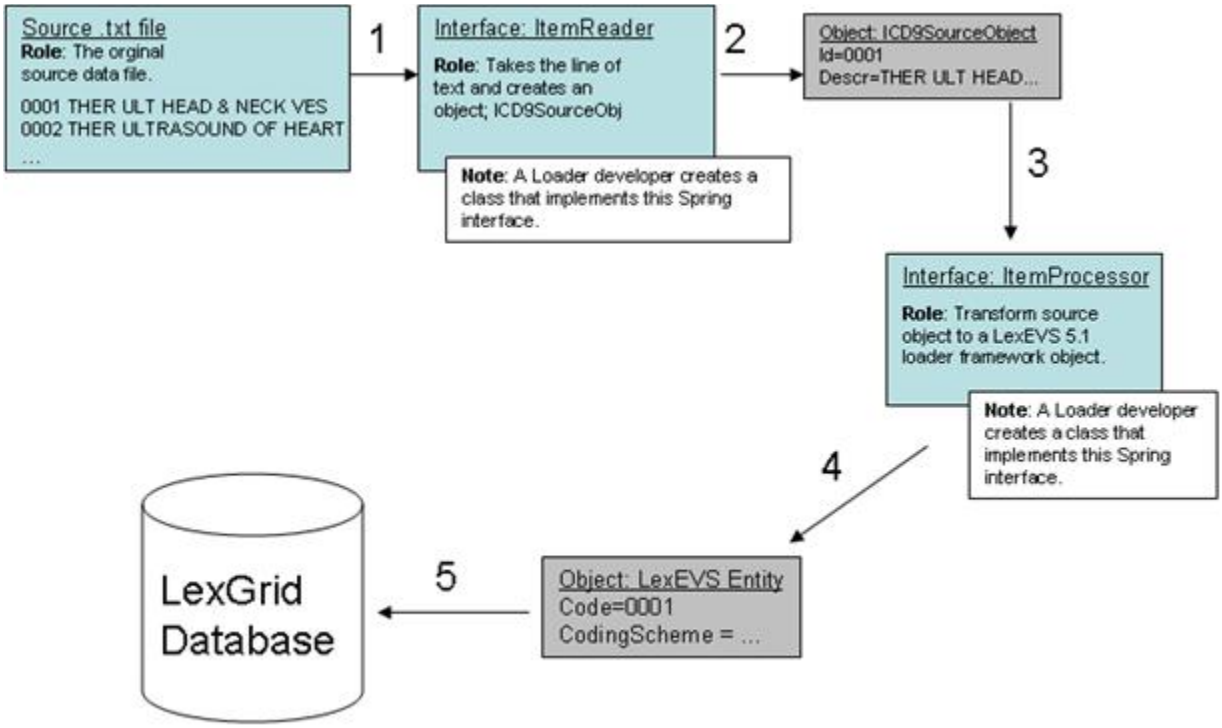
You can write a loader that uses the Loader Framework. The loader would follow this general process:

1. Read the raw data from the file into intermediate data structures, such as a user-defined ICD9SourceObject object.
2. Process the user-defined objects into LexGrid model objects.
3. Write the data in the LexGrid objects to the database.

An example may help in understanding the Framework. Our discussion will refer to the illustration below. Let's say we are writing a loader to load the ICD-9-CM codes and their descriptions, which are contained in a text file. We know we'll need a data structure to hold the data after we've read it so we have a class:

```
ICD9SourceObject {
String id;
String descr;
String getId() { return id; }
}
```

The Loader Framework uses Spring Batch to manage the reading, processing, and writing of data. Spring provides classes and interfaces to help do this work, and the Loader Framework also provides utilities to help loader developers. In our example, illustrated below, we will write a class that will use the Spring ItemReader interface. It will take a line of text and return an ICD9SourceObject (shown as 1 and 2). Next we'll want to process that data into a LexEVS object such as an Entity object. So we'll write class that implements Spring's ItemProcessor interface. It will take our ICD9SourceObject and output a LexEVS Entity object (shown as 3 and 4). Finally, we'll want to write the data to the database (shown as 5). Note that the LexEVS model objects provided in the Loader Framework are generated by Hibernate and utilize Hibernate to write the data to the database. This will free us from having to write SQL.



Spring

Configure Spring to be aware of your objects and to manage them. This is done via an XML configuration file. More details on the Spring config file arebelow.

ItemReader/ItemProcessor

Either write a class implementing this interface or use one of the Spring helper classes that already implement this interface. If you use one of the Spring classes, you may need to provide one of your own helper classes to construct your internal data structure object, such as ICD9SourceObject. Provide it to the Spring object via a setProperty call configured in the Spring config file.

Maven Set up

The projects containing the Loader Framework (**PersistanceLayer** , **loader-framework** , and **loader-framework-core**) use Maven for dependency management and build. You will still use Eclipse as your IDE and code repository, but you will need to install a Maven plugin for Eclipse.

1. Install the Maven plugin for Eclipse, which can be found at: [8].
2. Provide a URL and userid/password to a Maven repository on a server (which manages your dependencies or dependent jar files). Ours here at Mayo is: [9].
3. Import the Loader Framework classes from SVN.
4. You will most likely see build errors about missing jars. Resolve those by right clicking on the project with errors, select **Maven'**, and **Resolve Dependencies**. This will pull the dependant jars from the Maven repository into your local environment.
5. To build a Maven project, right click on the project, select **Maven**, then select

assembly:assembly.

Eclipse Project Set up

When you create a new loader project in Eclipse, it is recommended you follow the Maven directory structure. By following this convention, Maven can build the project and find the test cases.

The following diagram is from the Maven documentation:

Under this directory you will notice the following standard project structure.

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

The src/main/java directory contains the project source code, the src/test/java directory contains the test source, and the pom.xml is the project's Project Object Model, or POM.

For more information on the Maven project, see [10]

Configure your Spring Config (myLoader.xml)

Spring is a lightweight bean management container; among other things, it contains a batch function that is utilized by the Loader Framework. A loader using the framework will need to work closely with Spring Batch. The way it does that is through Spring's configuration file where you configure beans (your loader code) and how the loader code should be utilized by Spring Batch (by configuring a Job, Step, and other Spring Batch stuff in the spring config file). Here is sample code:

```
<job id="ioSampleJob">
   <step name="step1">
      <tasklet
         <chunk reader="fooReader" processor="fooProcessor" writer="compositeItemWriter" commit-interv
         </chunk>
      </tasklet>
   </step>
</job>

<bean id= "compositeItemWriter" class="...compositeItemWriter">
   <property name="delegate" ref="barWriter" />
</bean>

<bean id="barWriter" class="...barWriter" />
```

What follows is a brief overview of those tags related to the LoaderFramework. For more detail please see the Spring documentation at [11].

Beans

The **beans:beans** tag is the all-encompassing tag. You define all your other tags in it. You can also define an import within this tag to import an external Spring config file. (Import is not shown in the sample image above.)

Bean

Use these tags, **beans:bean'**, to define the beans to be managed by the Spring container by specifying the packaged qualified class name. You can also specify inititialization values and set bean properties within these tags.

```
<beans:bean id="umlsCuiPropertyProcessor" parent="umlsDefaultPropertyProcessor" class="org.lexgrid.loa
  <beans:property name="propertyResolver" ref="umlsCuiPropertyResolver" />
</beans:bean>
```

Job

The **job** tag is the main unit of work. The job is comprised of one or more steps that define the work to be done. Other advanced and interesting things can be done within the Job such as using **split** and **flow** tags to indicate work that can be done in parellel steps to improve performance.

```
<job id="umlsJob" restartable="true">
 <step id="populateStagingTable" next="loadHardcodedValues" parent="stagingTablePopulatorStepFactory"/>
...
```

Step

One or more **step** tags make up a job and can vary from simple to complex in content. Among other things, you can specify which step should be executed next.

Tasklet

You can do anything you want within a Tasklet, such as sending an email or a LexBIG function such as indexing. You are not limited to just database operations. The Spring documentation also has this to say about Tasklets:

```
The Tasklet is a simple interface that has one method, execute, which will be a called repeatedly
by the TaskletStep until it either returns RepeatStatus.FINISHED or throws an exception to signal
a failure. Each call to the Tasklet is wrapped in a transaction.
```

Chunk

Spring documentation says it best:

```
Spring Batch uses a "Chunk-Oriented" processing style within its most common implementation. Chunk-
oriented processing refers to reading the data one at a time, and creating "chunks" that will be
written out, within a transaction boundary. One item is read in from an ItemReader, handed to an
ItemWriter, and aggregated. Once the number of items read equals the commit interval, the entire
chunk is written out via the ItemWriter, and then the transaction is committed.
```

Reader

An attribute of the **chunk** tag. Here is the class that you defined implementing the Spring ItemReader interface to read data from your data file and create domain-specific objects.

Processor

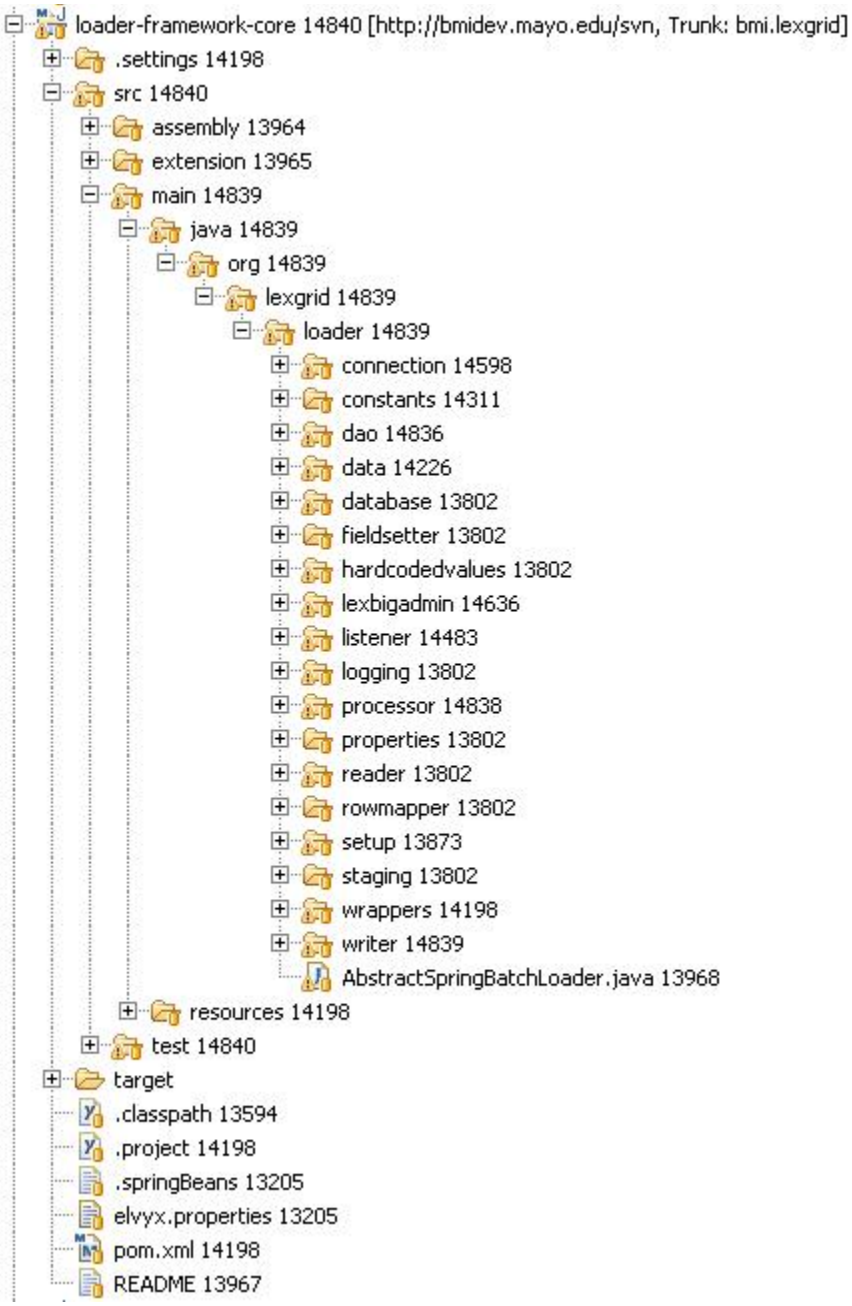
Another attribute of the **chunk** tag. This is the class that implements the ItemProcessor interface where other manipulations of the domain objects take place. In the case of the Loader Framework, we create LexGrid model objects from the domain objects so that they can be written to the database via Hibernate. Note that this is not a required attribute. In theory, if you had a data source from which you could read such that you could create LexBIG objects immediately, you would not need a processor. In practice this would most likely not be the case, but rather you need to work with the data to get it into LexBIG objects.

Writer

Attribute of the **chunk** tag. This class will implement the Spring interface ItemWriter. In the case of the Loader Framework, these classes have been written for you. They are the LexGrid model objects that use Hibernate to write to the database.

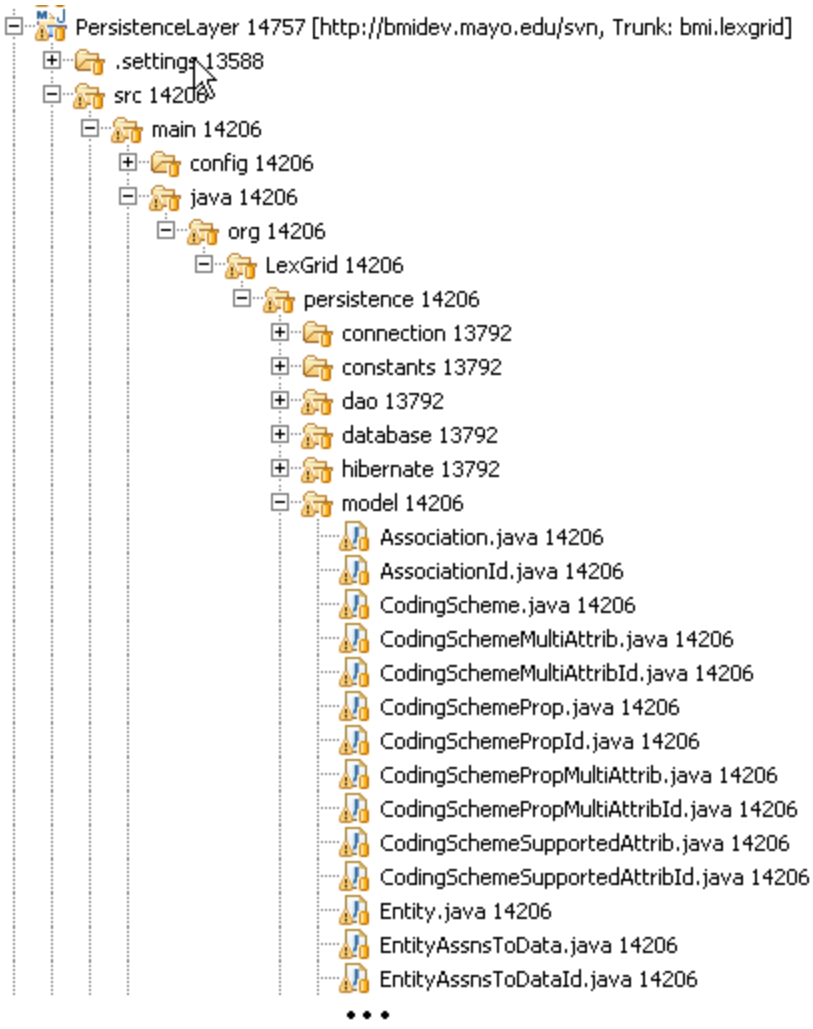
Key Directories

Below is an image of the loader-framework-core project in Eclipse, which shows the key directories of the Loader Framework. The following is a summary of the contents of those directories.



Directory	**Summary**
connection	Connect to LexBIG and do LexBIG tasks such as register and activate
constants	Assorted constants
dao	Access to the LexBIG database
data	Directly related to data going into the LexBIG database tables
database	Database-specific tasks not related to data, such as finding out the database type (MySQL, Oracle)
fieldsetter	Spring-related classes for helping to write to the database
lexbigadmin	Common tasks for LexBIG to perform, such as indexing

listener	Listeners you can attach to a load so that the code will execute at certain points in the load, such as a cleanup listener that runs when the load is finished, or a setup listener, etc.
logging	Access to the LexBIG logger
processor	*Important directory:* classes to which you can pass a domain-specific object and which will return a LexBIG object
properties	Code used internally by the Loader Framework
reader	Readers and reader-related tools for loader developers
rowmapper	Classes for reading from a database; currently experimental code
setup	*Loader developers should not need to dive into this directory.* Classes such as JobRepositoryManager that help Spring do its work; as Spring hums along it keeps tables of its internal workings.
staging	Helper classes to use if your loader needs to load data to the database temporarlily
wrappers	Helper classes and data strucutres such as a Code/CodingScheme class
writer	Miscellanous classes that write to the database. These are not the same classes you would use in your loader, i.e the LexBIG model objects that use Hibernate. Those classes are contained in the PersistanceLayer project (shown below). It is by using those classes in the PersistenceLayer that you let the Loader Framework do some of the heavy lifting for you.



Algorithms

None

Batch Processes

None

Error Handling

Spring Batch gives the Loader Framework some degree of recovery from errors. Like the other features of Spring, error handling is something you need to configure in the Spring config file. Basically, Spring will keep track of the steps it has executed and make note of any step that has failed. Those failed steps can be re-run at a later time. The Spring documentation provides additional information on this function. See [12] and [13].

Database Changes

None

Client

Loaders written to use the new framework will be called via the command line or script. Currently, the LexBIG GUI does not provide a framework to dynamically load extendable GUI components.

JSP/HTML

None

Servlet

None

Security Issues

None

Performance

Spring can accommodate parallel processing to enhance performance. The Spring documentation provides a good discussion of this topic. See [14].

Internationalization

Not internationalized

Installation / Packaging

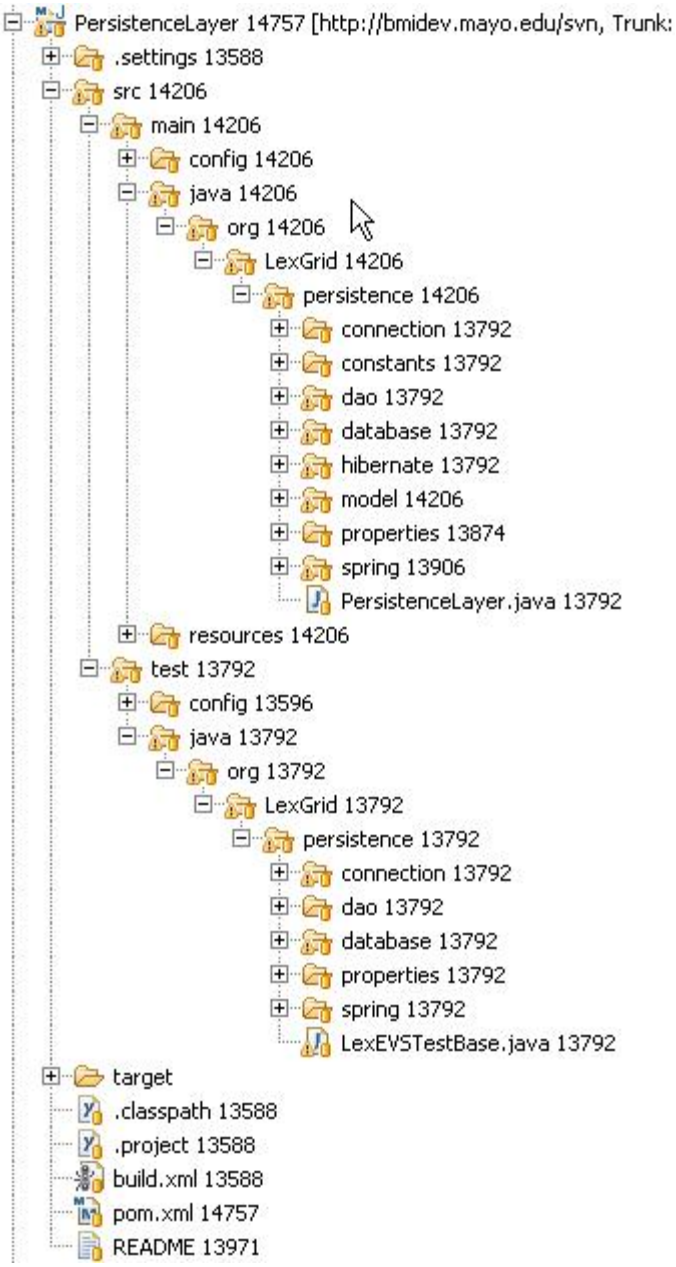
The Loader Framework is packaged as a LexBIG extension and thus is not included in the LexBIG jar

Migration

None

Testing

Automated tests are run via Maven. As mentioned earlier, the projects containing the Loader Framework code are configured to work with Maven. The illustration below shows the PersistenceLayer project and its standard Maven layout. Notice the structure of the test code mirrors the structure of the application code. To run the automated test in our Eclipse environment, we select the project, right click, select **Run As** and select **Maven test**. Maven does the rest.



Test Guidelines

The test cases are also integrated into the LexBIG 5.1 build environment and are run with each build.

Test Cases

See System Testing

Test Results

See System Testing

Custom Loader Feasibility Report and Recommendation

Persistence Layer Feasibility

The Persistence Layer enables LexEVS to have a single access point to the underlying database. This has several advantages:

- The DAO is implemented as an Interface, not a concrete class. We can implement this interface with Hibernate, JDBC, Ibatis, or any other Persistence tool or framework.
- All loaders can now share a single entry point to the database, and are not limited by memory constraints as some of the EMF persistence was.
- Connection Pooling and management is abstracted from the code and pluggable. Data source implementations may be switched and Connection Pooling may be configured without recompiling code.
- Transactions may be defined programmatically via AOP interceptors.

As LexEVS moves forward, the Persistence Layer is also flexible enough to play a part in the runtime Query API. With this, the runtime and loader code would be able to share a common Data Access Layer - we would then have a true DAO Layer.

Loader Framework Feasibilty

The Loader Framework has been implemented for two loaders: the UMLS single ontology loader and the NCI Metathesaurus loader. These loaders that implement the Loader Framework simply must define the READ and TRANSFORMATION mechanisms for the load, as well as load order and flow. All common details of loading to LexEVS will be dealt with by the Loader Framework and will not have to be implemented. Tools exist for:

- Lucene indexing
- registering CodingSchemes
- changing CodingScheme status (to ACTIVE, INACTIVE, etc)
- building the Transitivity Closure table
- adding supported attributes
- detecting database type
- staging temporary data to the database
- restarting failed loads
- integrating with LexEVS logging
- detecting and handling root nodes
- additional common LexEVS load-related tasks

Also, to aid in transformation, basic building blocks have been created that users may extend, such as:

- processors for all of LexEVS Model Objects
- various list processors
- grouping processors
- auto-supported attribute adding processors
- several basic resolvers to extract LexEVS Specific data from the source
- various other processors for specialized tasks

Several Utilities are also available for reading and writing, such as:

- group readers
- group writers
- writers configurable to skip certain records
- partitionable readers to break up large source files
- error-checking readers and Writers
- a validating framework for inspecting content before it is inserted into the database

Retrieved from "https://cabig-kc.nci.nih.gov/Vocab/KC/index.php/LexEVS_5.x_Loader_Framework"
Categories: VKC Contents | Documentation | LexEVS