

Reflections on the Design of Lightweight Java Persistence



Andrew Bowley

April 2015

Introduction

Design choices made in the creation of a useful software tool, in this case the Classy Tools lightweight Java persistence package, are good to capture and reflect upon as people try to understand what the package does and what is its applicability. I will go back to the first impulse which started the design process and explain what challenges presented themselves and how they were resolved.

Topics to be covered are Android, Object Relational Mapping (ORM), [OrmLite](#), Java Persistence Architecture (JPA), Dependency Injection, Entity Manager and heavyweight avoidance. I hope to take the reader on a journey into the thought processes of my mind and provide helpful insights into the decisions and compromises made along the way. One thing is certain, that after a considerable amount of effort bringing Classy Tools into existence, the design is pretty well fixed. I am pleased to say that I am happy with the results and confident that I have made a useful contribution to lightweight software development.

First Impulse

Android performance and stability demands provide the motivation for having lightweight application options available. The Android Developers "Core App Quality" criteria for Performance and Stability are:

Area	Description
Stability	App does not crash, force close, freeze, otherwise function abnormally on any targeted device.
Performance	App loads quickly or provides onscreen feedback to the user if the app takes longer than two seconds to load.
	With StrictMode enabled, no red flashes are visible when exercising the app.

Applications running on any platform benefit in stability and performance from having a lightweight approach, but how can this be achieved when working with relational databases? That was the question I faced when developing an Android application which stored a records management file plan on a local database.

Android comes with an SQLite database selected to suit a wide range of target systems, but what would be an appropriate ORM technology to sit on top of SQLite? The answer to that question came quickly.

I found that OrmLite Lightweight Java ORM was widely recommended as a package to use with Android. Quoting from the OrmLite web site, these features were consistent with a substantial ORM implementation:

- Setup your classes by simply adding Java annotations.
- Powerful abstract Database Access Object (DAO) classes.
- Flexible QueryBuilder to easily construct simple and complex queries.
- Supports MySQL, Postgres, Microsoft SQL Server, H2, Derby, HSQLDB, and Sqlite...
- Handles "compiled" SQL statements for repetitive query tasks.
- Supports "foreign" objects with the class field being the object but an id stored in the database table.
- Basic support for database transactions.

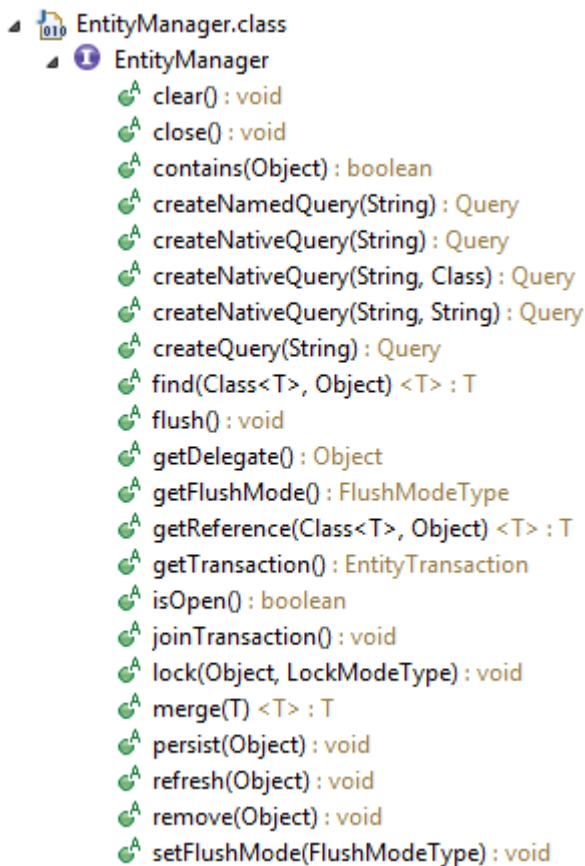
I was impressed that this was a portable package, supporting out of the box a number of diverse databases and having a pure Java core capable of "running everywhere". Using Java annotations to configure entity classes is a mature approach and support for Persistence API annotations is welcome. The DAO design is comprehensive and includes support for table creation. The QueryBuilder is also very useable with native SQL support provided to cover any gaps. It also supports "foreign" objects well and covers database transactions. So what did I find wanting?

Well configuration was the first thing that I felt needed improvement. OrmLite scans the classpath to find annotated classes requiring DAO support. This is not lightweight. A text configuration file can also be provided, but this has a non-standard format. Another concern was the "basic" support for database transactions, as this would require the copying of "boiler plate" transaction code in the absence of a system to support transactions. The solution seemed obvious, add some more Java Persistence API features to OrmLite: persistence.xml to configure "persistence units" and an EntityManager to provide a well-known transaction context. This was the start to Lightweight JPA.

Java Persistence Smorgasbord

The Java Persistence API is a standard which makes the developer's job easier when moving from one project to the next. It provides a familiar entry into diverse persistence implementations including Hibernate, Oracle JEE, EclipseLink and many more. It is also a contract to fulfil the guarantees implied by the operations being performed. For example `flush()` will "Synchronize the persistence context to the underlying database" and not result in any changes being uncommitted. OrmLite was not designed for JPA, so I needed to work out the implications of marrying the two together.

There are 2 versions, JPA1 and JPA2 and the first version being smaller is the best lightweight fit. OrmLite also references the annotation classes using the



version 1.0 Persistence API jar. I knew the key issue was how to implement EntityManager (left), which, as it's name suggests, does the heavy lifting on entity create, read, update and delete (CRUD).

Immediately I found two operations with no obvious means of support:

1. `lock()`
2. `joinTransaction()`

A lock is specified with a `LockModeType` which includes `READ`, `WRITE`, `OPTIMISTIC_READ`, `OPTIMISTIC_WRITE` etc. I checked but could not find reference to locking in OrmLite. I take it that there will usually only be one user accessing the database when running an application, and it is easy to add a version column for optimistic locking if there is a need.

The `joinTransaction()` method is for participation in distributed transactions which do not seem relevant to application execution.

There are also 5 query methods which also got me pondering as I felt a lightweight JPA would not accommodate scripted queries. Both Java Persistence query language and SQL, in scripted form, require compiling statements and transforming them into executable operations. Queries in coded form, on the other hand, are fine and are doable as named queries or through the Delegate object returned by `getDelegate()`.

There was also the issue of flushing. The EntityManager interface has a Flush-ModeType which controls whether changes made by a query are visible at the time of completion of a query. The default mode is supposed to be "AUTO" which means flushing takes place automatically. I was not sure what would be involved to get this to work so I took the easy "do nothing" option of having "COMMIT" as the only flush mode available.

At the end, I must admit to using JPA as a smorgasbord, selecting what was easy to do on top of OrmLite and avoiding what would be hard, if not impossible, to digest in a lightweight package. The JPA enthusiasts may be disappointed, but hopefully all is forgiven when we beat that 2 second response limit for user interactions.

Persistence Pays Off

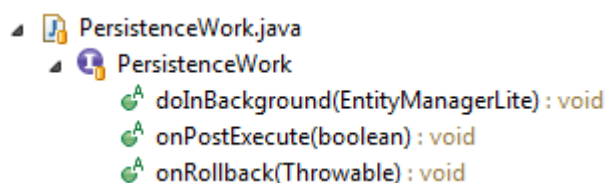
Next on the agenda was to work out how to achieve container managed persistence, in which the persistence context of any entity manager "is propagated along with any transaction that is currently active. If the transaction spans across components, all the entity manager object references that point to same persistence unit will have the same persistence context through out the transaction"¹. In EJB3, container managed persistence happens by "magic"., that is it happens by mostly invisible means in the application code. The only intrusion into the code is a @PersistenceContext annotation eg.

```
@PersistenceContext(unitName="actionBazaar")
private EntityManager entityManager;
```

The container injects the EntityManager reference and maintains it's persistence context in the background without having to do anything else. This is very elegant and avoids having to write tedious a potentially error-prone transaction code. I considered a lightweight approach would exclude any bytecode manipulation techniques or doing anything out of the ordinary. so it seemed there would be no chance of achieving the same elegance. The design I came up with has a PersistenceContainer class which instantiates the EntityManager object every time it is requested to perform work. This object is passed as a parameter as can be seen by looking at the **PersistenceWork** interface:

PersistenceWork has 3 methods:

- **doInBackground** executes in a background thread and passes an EntityManager object instantiated by the container.



```
PersistenceWork.java
PersistenceWork
doInBackground(EntityManagerLite) : void
onPostExecute(boolean) : void
onRollback(Throwable) : void
```

- **onPostExecute** runs in the caller's thread following successful completion.
- **onRollback** runs in the caller's thread if an error occurs.

As it turns out, **maybeExecuteInBackground** is a better name for the container task as I discovered that the single connection OrmLite JdbcConnectionSource does not support multi-threading, so the task, as of v1.0.3, runs synchronously in the caller's thread when any such ConnectionSource is detected.

Here is an example of `executeInBackground` for populating a new database with City objects:

```
@Override
public void doInBackground(EntityManagerLite entityManager)
{
    entityManager.persist(new City("bilene", 1718));
    entityManager.persist(new City("addis ababa", 8000));
    entityManager.persist(new City("denver", 5280));
    entityManager.persist(new City("flagstaff", 6970));
    entityManager.persist(new City("jacksonville", 8));
    entityManager.persist(new City("leadville", 10200));
    entityManager.persist(new City("madrid", 1305));
    entityManager.persist(new City("richmond", 19));
    entityManager.persist(new City("spokane", 1909));
    entityManager.persist(new City("wichita", 1305));
    // Database updates committed upon exit
}
```

By default PersistenceContainer runs in transaction mode and automatically commits on completion of the task. This can be changed by turning on user transaction mode. In this mode, the transaction object can be obtained from EntityManager and controlled by the program. However, any operation which may lead to a database change, such as persist, merge or remove an entity object is always placed in a new transaction if one is not currently active.

The EntityTransaction implementation is interesting in that it can, on it's own, act as a mini persistence container. The constructor can be passed a Callable object to be called just before commit happens. The persistence task is then invoked by simply calling `begin()` followed by `commit()`. The Callable is passed a DatabaseConnection object on which the transaction is pending, which allows database creation and update activities to be performed using SQL statements. By this means, all database transactions are performed in a consistent manner.

Classy Tools persistence does pay off by applying a standard API as far as the underlying OrmLite implementation and lightweight constraints allows, and then providing a persistence container and transaction implementation to avoid having to write error-prone transaction code.

Taking a Stab at Dagger

EJB3 @PersistenceContext annotation injects an object into an EntityManager reference and therefore decouples the persistence container from the application. Such decoupling, where the application is independent of the external systems it relies on makes code easy to test and maintain. When it comes to dependency injection, Spring is arguably the best known technology and now supports configuration by annotation as well as XML.

However, Spring works its magic at runtime and therefore not as strong a contender for lightweight work as newcomer Dagger, which employs the Java pre-compiler to reduce the runtime effort and also provides timely configuration error checking. The Dagger equivalent to EJB3 @PersistenceContext example is:

```
@inject @named="actionBazaar"
EntityManager entityManager;
```

The annotations @inject and @named are standard javax ones. The Dagger configuration is contained in one or more specially annotated classes that fabricate the objects to be injected. Here are the significant annotations:

Annotation	Description
@Module	Class annotation for a "module" which is a Dagger dependency object factory
@Provides	Annotates methods of a module to create a "provider" method binding for injecting dependent objects
@Singleton	A javax annotation to indicate only one instance of a dependent object is to be provided

Here is the module from the Many To Many sample:

```
@Module(injects = {
    WorkerRunnable.class,
    PersistenceFactory.class,
    NativeScriptDatabaseWork.class,
    PersistenceContext.class,
    DatabaseAdminImpl.class
})
public class ManyToManyModule implements ApplicationModule
{
    @Provides @Singleton ThreadHelper provideSystemEnvironment()
    {
        return new TestSystemEnvironment();
    }

    @Provides @Singleton ResourceEnvironment provideResourceEnvironment()
    {
        return new JavaTestResourceEnvironment("src/main/resources");
    }

    @Provides @Singleton PersistenceFactory providePersistenceModule()
    {
        return new PersistenceFactory(new SQLiteDatabaseSupport(ConnectionType.memory));
    }
}
```

The module identifies 5 classes with object dependencies to be satisfied by 3 singleton provider methods. The dependencies are all related to the system in which the application runs:

- **ThreadHelper** is an interface to set thread background priority —a dependency of WorkerRunnable class.
- **ResourceEnvironment** is an interface for streaming input sources such as files and also for getting the locale —a dependency of PersistenceFactory, amongst others, as it must read persistence.xml.
- **PersistenceFactory** is a concrete class which encapsulates the entire persistence context and for which only a single instance must exist. Note that a single DatabaseSupport constructor parameter determines both what type of database is used and whether it is in memory or on file.

As the module configuration is system-dependent, it cannot be hard coded. This prevents a package using dependency injection being “plug and play”. Its always “configure dependency, plug and play”. This is unfortunate, but is mitigated by the having the ability to nest modules inside each other, so once a configuration is found that suits the target system, it can then just be included in each application module from then on. Here is the ClassyFy Android module which includes an Android-specific module to satisfy all but one dependency:

```
@Module(injects = { ClassyFyStartup.class, ClassyFyProvider.class },
        includes = ClassyFyEnvironmentModule.class)
public class ClassyFyApplicationModule implements ApplicationModule
{
    @Provides @Singleton @Named(ClassyFyApplication.PU_NAME)
    AndroidPersistenceEnvironment provideAndroidPersistenceEnvironment()
    {
        return new AndroidPersistenceFactory()
            .getAndroidPersistenceEnvironment(ClassyFyApplication.PU_NAME);
    }
}
```

You can also see how @Named annotation can be applied to provide a specific implementation of an interface, here a context object for a particular persistence unit name.

Taking a stab at Dagger was a big risk at the start of a big project, but in the end, it turned out not only to work well in a lightweight package, but saved time by finding configuration errors early and thus avoiding a lot of run-fail testing. With dependency injection under control, I had to finalize the EntityManager design to determine if I could get away with a “plain old Java” to complete it.

Plain Old Java

Entities are naturally the focus of EntityManager. They are the objects in Object-to-Relational mapping and are distinguished by their classes being declared in persistence.xml. For each entity, OrmLite builds and caches a Data Access Object (DAO) for handling all aspects of entity life cycles. With OrmLite doing so much, the JPA EntityManager ends up as mostly a wrapper, imposing the JPA interface on top of the OrmLite implementation.

Entities are also Java beans with annotated setters and getters and otherwise “plain old Java” objects with no requirements regarding class inheritance or interface implementation. However, it is reasonable to require every entity class to have an identity (ID) field. as it provides a simple, consistent way to uniquely identify objects of the same type and build relationships. OrmLite DAOs have a generic ID parameter. ID is allowed to be declared of type Void or Object, in which case, the DAO `extractId()` method throws an exception when called. Lightweight JPA relies on every entity class having an ID field so it can use the `extractId()` method on the DAO to uniquely identify objects of the same type.

To hold all managed objects in a container in memory, Lightweight JPA has an **EntityKey** class which compounds entity type and ID. The management of objects is delegated to class **ObjectMonitor**.

```

ObjectMonitor.java
├── ObjectMonitor
│   ├── managedObjects
│   ├── removedObjects
│   ├── ObjectMonitor()
│   ├── getObjectsToUpdate() : List<Object>
│   ├── markForRemoval(Class<? extends Object>, Object) : void
│   ├── monitorNewEntity(Object, Object, Object) : boolean
│   ├── release() : void
│   └── startManagingEntity(Object, Object, PersistOp) <T> : T

```

ObjectMonitor has two containers, one for objects actively being managed and another for removed objects. The method `startManagingEntityObject()` is the entry point for EntityManager to synchronize objects passed to it with ObjectMonitor.

Note that there is a **PersistOp** parameter to identify what operation is about to be performed on the object and leads to some complicated logic relating to it's current state. PersistOp alternatives are:

- persist,
- merge,
- refresh,
- contains

The most interesting case is merge. What if an object to be merged has the same id as one already being managed? I decided the sensible thing to do is

swap out the new object for the managed object and then update the detached object so it is equivalent to the new object. In addition, a "dirty" flag is set on the EntityKey used to contain the new managed object so the database is updated on next transaction commit. Apache Commons BeanUtils copyProperties() utility method is used to align the merged and managed objects. This may be slow due to a reliance on reflection, but it works without any fuss. Anything referencing the pre-merge managed object will read the correct values, that is, unless another merge takes place, but I thought that would be unlikely.

In the end, plain old Java, was able to perform the task of managing persistent objects with one constraint about making an ID field mandatory. One other significant caveat to keep in mind is that operations performed by queries are not monitored and it is possible to subvert EntityManager by making changes directly to the database eg. delete a row mapped to a managed entity. However, it is envisaged that queries will mostly perform search operations which return entity IDs.

We are now at where I intended to wind up with my reflections. I wanted to use JPA in my applications, including those which run on Android. That goal has proven doable while having to face some very interesting challenges, some of which I have shared in this article. The following table provides an overview of Lightweight JPA behaviour which covers the full entity life-cycle. For more information, including links to the software downloads, go to [Cybersearch2 web site](#).

pto...

Lightweight JPA Overview

Event	Activity
At start up, application calls <code>initializeAllDatabases()</code> on <code>PersistenceContext</code> object.	Create database if it does not exist. Persistence unit (PU) contains relevant properties such as database name. For each entity class defined in the PU, set up <code>OrmLite</code> DAO configuration and create entity table, if necessary.
Persist entity object	Use <code>OrmLite</code> DAO <code>create()</code> to save the object to the database and then add it to the container of managed objects.
Merge entity object	Add to managed objects or, if already managed, swap and align new and managed objects. Also set "dirty" flag on container key to indicate the database must be updated on commit.
Refresh entity object	Use <code>OrmLite</code> DAO <code>refresh()</code> to fetch current database version to the managed object.
Remove entity object	Use <code>OrmLite</code> DAO <code>delete()</code> to delete the entity in the database. The previously managed object is transferred to the removed objects container so an error can be flagged if any further references are made to it.
Find or get reference to entity by primary key	Use <code>OrmLite</code> DAO <code>queryForId()</code> to fetch and return the specified object. Note that there is no change to transaction state with this operation. The difference between find and get reference is, when the entity is not found, the former returns null and the latter throws an exception.
Flush <code>EntityManager</code>	Commits current transaction and begins new one.
Clear <code>EntityManager</code>	Rolls back current transaction, clears managed objects container and begins new transaction.
Close <code>EntityManager</code>	Commits current transaction and clears managed objects container.
Test database contains entity	If entity object is managed, return true, else return result of <code>OrmLite</code> DAO <code>entityExists()</code> .
Get Transaction	If not in user transaction mode, returns proxy for which only <code>setRollbackOnly()</code> is active.