App Engine JDO vs. JPA

Paul Wein, *university of applied sciences regensburg*

This paper describes the differences of using Java Persistence API (JPA) and Java Data Objects (JDO) compared to the usage of the proprietary datastore API.

Introduction

The Google App Engine datastore offers different ways to persist your data. This paper will compare different approaches and talk about advantages and disadvantages of each technique.

For better understanding and a real practical point of view, I developed a web application that uses this different techniques. The main function of this application enables users to sign in with their google accounts and create and manage notes. These notes can be managed in lists.

For testing the different persistent APIs the persistence layer needs to be strictly seperated from the application logic. The design concept of the data access object was implemented for this purpose, seperating the different ways the notes are persisted within the datastore. Each persistence mechanism has its own implementation and is abstracted through interfaces and a factory that holds and provides the implementations.

The user can decide within the application settings which persistence mechanism he wants to use. All data access objects read and write to the same datastore. Unless performance the user won't notice any differences.

Persistence Frameworks

Many developers are used to persistence frameworks which make the process of object-relational mapping more easy. The persistence framework manages the database and the mapping between the database and the objects There are two standardised persistence technologies available for Java. JDO which was released in may 2001 and JPA which was released in may 2006. Those specifications enable you to interact with databases without using any database-specific code. This simplifies the development process and makes your software more flexible.

JDO (Java Data Objects)

Java Data Objects is a standard interface for storing Java Objects in a transparent way. The application code won't contain any database-specific code, which makes it much easier for the developer to exchange the underlying datastore. This interface can be used to store Java Objects to relational databases, object databases, XML or any datastore. The App Engine uses an open source implementation called DataNucleus Access Plattform for the App Engine Datastore. As you can see in Figure 1 JDO does not define the type of datastore. [3]

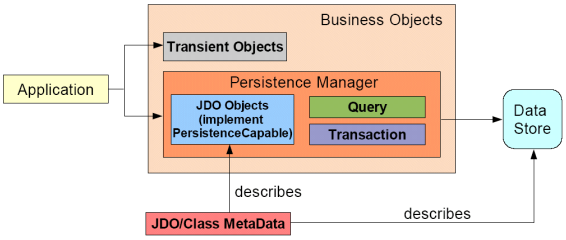

Figure 1: *JDO Architecture*

JPA (Java Persistence API)

The Java Persistance API is a java programming language framework. It is similar to JDO and shares similar roots. The Java Persistence API was developed in part to unify the Java Data Objects API, and the EJB 2.0 Container Managed Persistence (CMP) API. [4] JPA specifies only relational databases. Google App Engine uses equal to JDO the open source implementation DataNucleus Access Plattform. The App Engine SDK contains an Access Plattform Adapter which uses the low-level datastore API. There are some restrictions in the usage of the JPA API with the datastore. JPA was designed for relational databases and global transactions. The datastore which is a NoSQL database has no posibility to join 'tables' and transactions are limited to entity groups. [5]

Proprietary Datastore API

The proprietary Datastore API is used by the DataNucleus Access Plattform Adapters for JPA and JDO, but can also be used by the application developer. The datastore holds dataobjects known as entities. You can persist an retrieve entities with the low-level API. You can not persist or retrieve your java objects directly because datastore only knows entities. You need your own mapping mechanism to translate persistent objects to your business logic objects. This is a lot of more code to implement than using frameworks with relational object mappers included, but gives the developer more flexibility in his schemas.

Personal Assistant

The web application personal assistant allows users to create and manage notes and lists of notes. The users can sign in with their google account and create, update, delete or search for their notes. It was built to understand the differences between the former mentioned persistence techniques. For this reason the user has the possibility to change the persistence mechanism within the settings page of the web application.

Architecture

Personal Assistant is written in java using servlets and java server pages. There exists one template that contains the main menu and the searchbar and all other dynamic contents are injected. [6, 7]

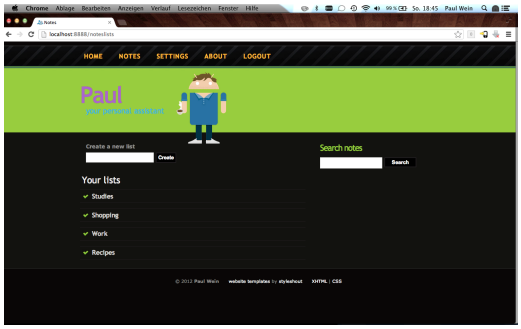

Figure 2: *Personal assistant*

To be able to switch the persistance mechanism at runtime the persistence layer is completely seperated from the business logic.

The data access object software design pattern was used to reach that goal. An interface defines the desired operations and each persistence technique has its own implementation of this interface. The factory contains a hash map with the implementations and returns the desired implementation depending on the current settings selected by the user. In Figure 3 you can see the architecture of the persistence layer.

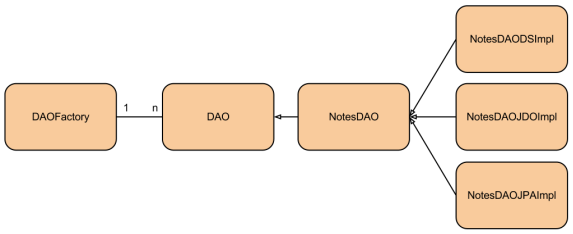

Figure 3: *DAO Architecture*

JPA and JDO are also used to abstract from the underlying datastore. But in this particular case also JPA and JDO need to be seperated to be able to be exchanged.

Website & Servlets

In Figure 4 you can see the flow of the website. Each box is a servlet and a corresponding jsp page.

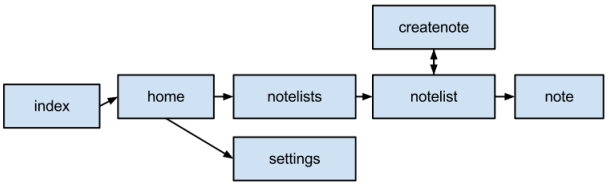

Figure 4: *Website Architecture*

The base class of every servlet used in this application is NotesAppServlet. Its handling the authorization process and contains the user attribute which is

accessible from every servlet that extends this servlet. It also contains methods that redirect to an error page in error cases. If a user is navigating to a page without being signed in, the NotesAppServlet will redirect the user to the sign in page. After signing in from the index page the user is forwarded to the home page. From there on he is able to navigate through the pages until he signs out again. If the user is already signed in, the index page forwads him directly to the home page.

Every Servlet is interacting with the model and filling the response attributes. The jsp pages are using taglibs for displaying the data.

POJO (Plain Old Java Object)

The Note class contains the following persistent attributes:

- Key key
- Date date
- String subject
- String note
- String userId

The NoteList class contains the following persistent attributes:

- Key key
- String name
- List<Note> notes
- String userId

As you can see the userId is redundant. The reason for that is, that there is no possibility to join those 'tables'. I also implemented a search function that searches for notes across all note lists. So if you can not join these 'tables' there is no easy way to find out which note belongs to which user if there is no userId in the note object. One approach would have been fetching all note lists of the user and performing ancestor queries on them. But I think it is more efficient querying index based over all notes and finding the related ones. Otherwise you would have to perform one query per note list. Also using ancestor queries is not possible using JDO or JPA. You would have to add a parent property to each child entity, which would allow you to query on the immediate parent of an entity. Adding this redudant userId property is no problem, because notes and note lists can only be accessed via the same user, so it is not possible to modify these properties with different userIds and they will stay consistent.

JDO and JPA are using java reflections for examining these classes and persisting these attributes. To tell these frameworks which attribute needs to be persisted and which attribute is the primary key you can use annotations. It is possible to use JDO and JPA simultaneously. This makes it possible to use the exact same classes for the different persistence frameworks. Listing 1 shows an example of defining the primary key with the different annotations.

Listing 1: *JDO/JPA annotations example*

```
// JPA Annotations
@Id
@GeneratedValue(strategy =
    GenerationType.IDENTITY)

// JDO Annotations
@PrimaryKey
@Persistent(valueStrategy =
    IdGeneratorStrategy.IDENTITY)
private Key key;
```

Persistence Layer

The persistence layer consists of the DAOFactory object which is returning the desired implementation depending on the current user setting. To overcome the chicken or the egg problem the user settings are always saved and loaded with the proprietary datastore API. Every user gets their own settings entity containing an integer property which defines the datasource. The DAOFactory returns the implementations stored in a HashMap depending on the integer property.

The following generic method returns the specified Implementation of the DAO interface.

```
public <T extends DAO> T getDAO(
        Class<?> requiredDao,
        Class<T> daoClass )
```

There is an interface called NotesDAO which defines the operations for persisting and retrieving notes or note lists. There also exist three implementations of this interface for the corresponding persistence technique.

- NotesDAODSImpl (proprietary datastore API)
- NotesDAOJDOImpl (JDO)
- NotesDAOJPAImpl (JPA)

The JDO and JPA implementations work directly with the Note and NoteList classes. The datastore API is not able to persist java objects directly, so I implemented a transformer class which translates between entities and the corresponding objects. Also the same classes are used within this transformer class, so there exists only one implementation of these classes. Otherwise the DAO Pattern would be more difficult to realize.

Relationships

The first step for defining the data model of my application was to find out how JDO and JPA were storing the objects. Luckily the relationships are equal. The NoteList object, which contains a list of Note objects, is the parent of the entity group. Each note is created as a child of the parent note list. The NoteList entity contains a multi valued property that is containing the keys of note entities in that list. You can see the data structure of notes and note lists in Figure 5.

ID/Name	name	notes	userId
id=19001	Shopping	[datastore_types.Key.from_path(u'NotesList', 19001L, u'Note', 1L, _app=u's~notizenapp'), datastore_types.Key.from_path(u'NotesList', 19001L, u'Note', 1001L, _app=u's~notizenapp')]	11223682001942184991 6
id=20001	Studies	[datastore_types.Key.from_path(u'NotesList', 20001L, u'Note', 1L, _app=u's~notizenapp')]	11223682001942184991 6

ID/Name	date	note	subject	userId
id=1	2012-12-26 10:37:16.884000	tomatoes, flour, cheese, rucola	pizza ingredients	11223682001942184991 6
id=1001	2012-12-26 10:37:39.193000	cola and orange juice	drinks	11223682001942184991 6
id=1	2012-12-26 10:38:14.644000	finish the paper, create a video	data in the cloud	11223682001942184991 6

Figure 5: *note list & note data structures*

So now after I knew how JDO and JPA structure their data I needed to implement the same data structure for the proprietary datastore API. To reach the flexibility to exchange the persistence mechanism all implementations need to rely on the same schema of the data. Without that requirement it would not have been necessary to store the notes keys as a multi valued property in the note list entity. You can also retrieve the notes with an ancestor query. If I only had used the datastore API my data schema would look a little bit different. I would not use the unowned relationship from note list to note with the notes keys. Because doing so makes the implementation of the CRUD operations more complex, because you always have to operate on top of your entity group. So with entity groups and ancestor queries you can retrieve all notes from one note list and you don't have to modify any of the note list properties if you modify your note. The relationship is only given because of the note list key is used to create the note key. If you delete one note, this does not affect any of the note list properties. If you use JDO or JPA your schema is given and you have to load your note list and delete the note, because you also need to delete the note key of the multi valued note list property.

Differences

Using JDO and JPA is very similar. The annotations sound very similar and the process and methods of the CRUD methods are also nearly the same. The so called 'Persistence Manager' of JDO is called 'Entity Manager' in JPA. [1] The main differences are found in the query languages. The JDO query language (JDOQL), which is based on java syntax, differs from the JPA query language (JPQL), which is based on SQL syntax.

When using the prorpietary datastore API you have to give a name for each property of the entities. First I used the upper case names 'Note' and 'Subject' but the persistence frameworks were giving the lower case names. So the schema was not the same and the logic did not work. One solution would have been to use the @Column annotation and give the upper case names. But I decided to just rename my propertyname constants.

Scalability

So does this architecture work in terms of scalability? The answer is yes. Because every user has it's own note list entities, which are the parents of the entity groups. So multiple transactions from different users are possible at the same time, because they work on different entity groups. It would also be possible to modify more than one note from the same note list at a time, but this is not neccessary because the work flow of this application is different. Users only modify one single note at a time, but multiple users can modify their notes without interfering each other.

Think about it

It feels like the perfomance is much better without the persistence frameworks. JPA and JDO are doing a lot of extra checks and are adding some overhead to the datastore API. Also your schemas are not that flexible. On the other hand it is much faster to use while setting up your application, because you do not have to write your own object mappers. You also might have less errors, because these frameworks force your application logic to use this standardized way of persistence and the framework does loading and persisting for you.

Summary

I think using these persistence frameworks is a nice way to migrate existing apps to the Google app engine. But it is kind a tricky to use a framework like JPA that was designed for relational databases with a NoSQL database. If you do not have any existing application that needs to be migrated to app engine and you do not need to port to any other plattform I would recommend using a framework like Objectify that was designed for the datastore.

One benefit of these persistence frameworks was that they also handle the creation of database tables. As we know the NoSQL datastore has no fixed schema and also no tables. So you can store your entities without the need to create a fixed data schema. The benefit of automatic database creation is no real benefit in a NoSQL database.

You may want to start a new project on google app engine with jdo or jpa for portability reasons. Than the decision which framework to choose is a decision of flavour. As you can see in the google search trend analyis of Figure 6 jdo and jpa are nearly equal. [2] Maybe jdo is a little bit on top because the google app engine documentation tells you more about jdo than jpa. Also jdo is a datastore-agnostic approach to object persistence while jpa aims relational databases.

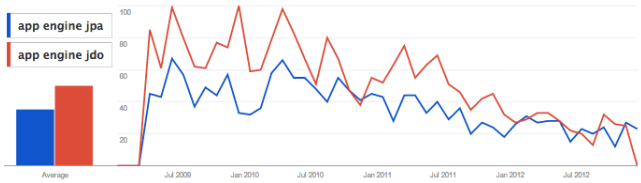

Figure 6: *app engine jpa vs. app engine jdo*

Another reason for choosing jdo or jpa could also be that the developer is already used to it. This makes the development process much faster, because the developer knows how these frameworks work and can write a usual web application on google app engine. But he should of course be aware of the design principle of entity groups and transactions to be able to write a scalable application.

As you can see there are several reasons for choosing these frameworks but also some for not choosing them. It depends on the developers knowledge but also on the application requirements. Personal assistant shows you that it is possible using these frameworks, but as I mentioned above I would also use a different data structure if I had not used them. So mostly I would say it is a design decision.

References

[1] Apache Software Foundation. Jdo .v. jpa : Api. `http://db.apache.org/jdo/jdo_v_jpa_api.html`.

[2] Google. Search trends. `http://www.google.com/trends/explore#q=app%20engine%20jpa%2C%20app%20engine%20jdo&date=1%2F2009%2048m&cmpt=q`, December 2012.

[3] Qusay H. Mahmoud. Getting started with java data objects (jdo). `http://192.9.162.55/developer/technicalArticles/J2SE/jdo`, August 2005.

[4] John O'Conner. Using the java persistence api in desktop applications. `http://192.9.162.55/developer/technicalArticles/J2SE/Desktop/persistenceapi`, June 2007.

[5] Dan Sanderson. *Programming Google App Engine*. O'Reilly, first edition edition, November 2009.

[6] Jorge Simão. View templates with jsp. `http://www.jpalace.org/docs/articles/jsp-layout/jsp-layout.html`, 2011.

[7] Styleshout. Css template. `http://www.styleshout.com/templates/preview/NewHorizon11/index.html`, December 2010.