App Engine ORM User Guide

Introduction

App Engine ORM provides transparent persistence of your application's model objects using your choice of JDO or JPA, both of which are standard java persistence apis. The bulk of the implementation of these standards comes from an open source product called DataNucleus. DataNucleus has a plugin model that enables extensions for other types of datastores. The App Engine ORM is a DataNucleus plugin that adds support for the App Engine Datastore. The goal of this document is to describe how to configure your Java App Engine application to use App Engine ORM. This document does **not** teach you how to use JDO or JPA. App Engine ORM is an implementation of the JDO and JPA specifications, so you should assume that the behavior and api of App Engine ORM is consistent with these specifications unless explicitly called out in this guide.

What's Inside appengine-orm.zip?

File or Directory	Description
asm-3.1.jar	Byte code manipulation library used by DataNuclues
datanucleus-core-1.1.0.m1-src.zip	Source for DataNucleus core
datanucleus-enhancer-1.0.1-src.zip	Source DataNuclues enhancer
datanucleus-enhancer-1.0.1.jar	DataNucleus enhancer
datanucleus-jpa-1.1.0.m1-src.zip	Source for DataNucleus JPA support
datanucleus-jpa-1.1.0.m1.jar	DataNucleus JPA support
datanucleus-rdbms-1.1.0.m1-src.zip	Source for DataNucleus rdbms support
datanucleus-rdbms-1.1.0.m1.jar	DataNucleus rdbms support. Even though the App Engine datastore is not a relational database, we do make use of some of DataNucleus' relational database related features.
jdo2-api-2.2.jar	JDO api
jta.jar	Java Transaction API
persistence-api-1.0.2.jar	Java Persistence API
datanucleus-core-1.1.0.m1-000X.jar	DataNuclues core, currently split up across multiple jars to get around some file size limitations.
datanucleus-appengine-0.4.jar	DataNucleus AppEngine Datastore plugin
datanucleus-appengine-0.4-src.zip	DataNucleus AppEngine Datastore plugin sources
README	Release Notes
demos (directory)	Contains ORM demo applications. Update demos/demos.properties to point to a directory containing appengine-api.jar and appengine-local-runtime.jar and then run 'ant' in any

	subdirectory to build and launch that application using the dev appserver.
docs	App Engine ORM documentation

Enhancing Your Classes

App Engine ORM makes your POJOs transparently persistable using the DataNucleus bytecode enhancer. If you're just running your app in the dev appserver you can enable runtime bytecode enhancement by adding the following jvm flag:

```
-javaagent:lib/datanucleus-
enhancer-1.0-SNAPSHOT.jar=package.with.classes.needing.enhancement
```

You'll pay a performance penalty, but hey, it's just the dev appserver. No big deal, right?

This is certainly the fastest way to get your app up and running, but you'll need to enhance your actual classfiles before you deploy your code to Google's servers. Fortunately, DataNucleus includes an ant task that makes this pretty straightforward. Here's a snippet from demos/helloorm/build.xml that performs enhancement:

```
<path id="enhancer.classpath">
  <pathelement location="${orm.lib.dir}/jdo2-api-2.2-SNAPSHOT.jar"/>
  <pathelement location="persistence-api-1.0.2.jar"/>
  <pathelement location="${orm.lib.dir}/datanucleus-enhancer-1.0-SNAPSHOT.jar"/>
  <pathelement location="${orm.lib.dir}/datanucleus-core-1.1-SNAPSHOT-0000.jar"/>
  <pathelement location="${orm.lib.dir}/datanucleus-core-1.1-SNAPSHOT-0001.jar"/>
  <pathelement location="${orm.lib.dir}/datanucleus-core-1.1-SNAPSHOT-0002.jar"/>
  <pathelement location="${orm.lib.dir}/datanucleus-core-1.1-SNAPSHOT-0003.jar"/>
  <pathelement location="${orm.lib.dir}/datanucleus-jpa-1.1-SNAPSHOT.jar"/>
  <pathelement location="${orm.lib.dir}/datanucleus-rdbms-1.1-SNAPSHOT.jar"/>
  <pathelement location="${orm.lib.dir}/asm-3.1.jar"/>
  <pathelement location="classes"/>
</path>

<target name="enhance" description="enhance" depends="compile">
  <taskdef name="datanucleusenhancer" classpathref="enhancer.classpath"
        classname="org.datanucleus.enhancer.tools.EnhancerTask"/>

  <datanucleusenhancer classpathref="enhancer.classpath" failonerror="true">
    <fileset dir="classes">
      <include name="**/*.class"/>
    </fileset>
  </datanucleusenhancer>
</target>
```

JPA Guide

Configuration

Create a persistence.xml file. Here's a sample persistence.xml for App Engine ORM, taken from demos/helloorm/src/META-INF/persistence.xml:

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
       http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

    <persistence-unit name="helloorm">
       <class>com.google.appengine.helloorm.Flight</class>
       <properties>
          <property name="datanucleus.NontransactionalRead" value="true"/>
          <property name="datanucleus.NontransactionalWrite" value="true"/>
          <property name="datanucleus.ConnectionURL" value="appengine"/>
       </properties>
    </persistence-unit>
<persistence>
```

And then in your code...

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("helloorm");
// off to the races
```

JDO Guide

Configuration

Create a datanucleus.properties file and place it at the root of your classpath. Here's a sample datanucleus.properties for App Engine ORM, taken from demos/helloorm/src/datanucleus.properties:

```
javax.jdo.PersistenceManagerFactoryClass=org.datanucleus.jdo.JDOPersistenceManagerFactory
javax.jdo.option.ConnectionURL=appengine
datanucleus.NontransactionalRead=true
datanucleus.NontransactionalWrite=true
```

And then in your code...

```
PersistenceManagerFactory pmf =
JDOHelper.getPersistenceManagerFactory("datanucleus.properties");
// off to the races
```

Primary Keys

Every POJO that is mapped to the datastore must have a single member as its primary key. The type of this member must be java.lang.String or com.google.apphosting.api.datastore.Key. The value of this member will correspond to a unique identifier for an Entity in the App Engine Datastore, which consists of your application id, the kind of the entity, and an additional value provided either by the datastore or by the user. If you want the datastore to provide this value for you just leave your primary key field null. App Engine ORM will treat this as a request for the datastore to assign an id during the process of making your POJO persistent. If you want to provide this value yourself, you'll need to do something slightly different depending on the type of your primary key member. If your primary key member is a String, assign a String-ized representation of the value you want to provide directly to the primary key member. If your primary key member is a Key, the story is a bit uglier. You'll need to assign a Key instance to your Key primary key member, but in order to create this Key instance you'll need to invoke

```
com.google.apphosting.api.datastore.KeyFactory.createKey("ignored", "my
value");
```

If you read the javadoc for the createKey() method you'll see that the first argument is the Entity kind and the second argument is the user-provided component of the key. There is no way to create a Key with only the user-provided component, so unfortunately, in order to make this work for App Engine ORM, we need to create a Key with a dummy kind ("ignored"). It does not matter what dummy kind you provide so long as it as a non-null, non-empty String. The value will be ignored - the kind of the entity that is created will be derived from the class of your POJO, not from this Key. If you find yourself creating keys with dummy kinds frequently, consider creating a helper method along the lines of

```
Key createPrimaryKey(String val) {
    return KeyFactory.createKey("ignored", val);
}
```

This will at least hide some of the ugliness.

Whether your primary key is a String or a Key, and whether you choose to provide a value for the primary key or let the datastore assign it, App Engine ORM will modify the primary key value during the process of making your object persistent. If your primary key is a Key, the primary key field's value will be set to a fresh Key instance with its Kind property set and either its Id or Name property set (one or the other but never both). Id will be set if you let the datastore assign the id. Name will be set if you provided the id yourself. If your primary key is a String, the primary key field's value will be set to a String encoding of a Key object. This String encoding is simply the result of calling KeyFactory.encodeKey(key), and you can access the various components of the primary key using

```
Key pk = KeyFactory.decodeKey(pkStr);
pk.getKind();
pk.getName();
pk.getId();
```

String or Key?

Deciding whether to model your primary keys on your POJOs as Strings or Keys is an important design decision. How do you choose?

The recommendation at this point (still early) is to use Strings unless you have a very strong reason not to. Here's why:

- Portability - Using Key brings in a compile-time dependency on App Engine, and considering one of the main goals of App Engine ORM is to provide standards-based persistence, that's really a shame. If you use Key you can't just pick up your code and run it in another container. Using String for your primary keys doesn't automatically grant you portability, but it certainly doesn't close the door the way using Key does.
- Cleaner semantics. Setting a user-defined id on a POJO with a String primary key is straightforward and intuitive. Setting a user-defined id on a POJO with a Key primary key involves creating a hacky instance of a Key with a dummy kind.
- With JPA, you can't have a non-pk field of type Key. That means if you're using JPA there's no way to model your ancestor pk using Key. Having an ancestor pk

of type String an a pk of type Key works, but it's ugly. It's cleaner to use String for both.

So those are the reasons in support of using String. Does it ever make sense to use Key? If you frequently need access to the individual components of your primary key (kind, id/name), you may be better off using Key instead of String because it will save you the trouble of having to convert the String into a Key in order to access these individual components. There's nothing preventing you from exposing accessors for these individual components on your POJO and doing the converson on-demand, so the recommendation would still be to use String and try this out. If and only if you have conclusive evidence that these conversions are making your app too slow to use, consider switching from String to Key.

Inserting POJOs With Ancestors

When you persist a POJO to the datastore, one of the things you get to choose is which persistent object key, if any, is the parent of the POJO you are persisting. App Engine ORM supports this datastore feature through a JPA/JDO extension. Here's an example for JPA:

```
@Entity
public class HasAncestorJPA {

  @Extension(vendorName="datanucleus", key="ancestor-pk", value="true")
  private String ancestorId;

  @Id
  @GeneratedValue(strategy= GenerationType.IDENTITY)
  private String id;

  public HasAncestorJPA(String ancestorId) {
    this(ancestorId, null);
  }
 }
```

and here's the corresponding example for JDO:

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class HasAncestorJDO {
  @Persistent
  @Extension(vendorName="datanucleus", key="ancestor-pk", value="true")
  private String ancestorId;

  @PrimaryKey
  @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
  private String id;

  public HasAncestorJDO(String ancestorId) {
    this(ancestorId, null);
  }
}
```

Adding the Extension annotation to a field tells App Engine ORM to use the value of this field as the parent key when it creates the persistent Entity. This field must be a String or a Key. Note that creating a field for the ancestor and annotating it as above is not a

commitment to providing an ancestor for every POJO of this type - null is a perfectly valid value for this field.

Querying POJOs By Ancestor

The datastore can fulfill ancestor queries. Even though ancestors are not part of the JPA or JDO specifications, you can issue JPQL and JDOQL queries that restrict by ancestor without any special syntax.

In JPQL:

```
    Query q = entityMgr.createQuery(
        "select from " + HasAncestor.class.getName() + " where ancestorId =
:ancId");
    q.setParameter("ancId", ancestorKeyStr);
    return q.getResultList();
```

In JDOQL:

```
    Query q = persistenceMgr.newQuery(
        "select from " + HasAncestorJDO.class.getName() + " where
ancestorId == ancId parameters String ancId");
    return q.execute(ancestorKeyStr);
```

Query Limitations

Although JDOQL and JPQL are feature-rich query languages, App Engine ORM only supports the features of these query languages that can be natively fulfilled by the datastore. Please read http://code.google.com/appengine/docs/datastore/queryclass.html for information on the types of queries supported by the datastore. You should be able to conceptually map the datastore query capabilities to the corresponding features in JDOQL and JPQL, but here's a summary:

- You can have as many equals filters as you want as long as they're separated by AND.
- You can have exactly one inequality filter (>, >=, <=, <).
- No aggregation functions (min, max, avg, etc.).
- No joins.

Transactions

You'll need to read and understand http://code.google.com/appengine/docs/datastore/transactions.html before you can make proper user of transactions with App Engine ORM. While we support standard transaction management apis, the semantics are probably a bit different from what you're used to because the App Engine Datastore does not support global transactions. More information to be added by Erick.....