Google App Engine for Java, Part 3: Persistence and relationships

Java-based persistence and the Google App Engine datastore

Richard Hightower
August 25, 2009

Data persistence is a cornerstone of scalable application delivery in enterprise environments. In this final article of his series introducing Google App Engine for Java™, Rick Hightower takes on the challenges of App Engine's current Java-based persistence framework. Learn the nuts and bolts of why Java persistence in the current preview release isn't quite ready for prime time, while also getting a working demonstration of what you can do to persist data in App Engine for Java applications. Note that you will need to have the contact-management application from Part 2 up and running as you learn how to use the JDO API to persist, query, update, and delete Contact objects.

View more content in this series

App Engine for Java seeks to take the worry out of writing a persistence layer for scalable Web applications, but how well does it achieve that aim? In this article, I conclude my introduction to App Engine for Java with an overview of its persistence framework, which is based on Java Data Objects (JDO) and Java Persistence API (JPA). While initially promising, App Engine's Java-based persistence currently has some serious drawbacks, which I explain and demonstrate. You'll learn how App Engine for Java persistence works, what the challenges are, and also what persistence options you have when working with Google's cloud platform for Java developers.

As you read the article and work through the examples, you'll want to keep in mind the fact that the App Engine for Java is currently a preview release. While the Java-based persistence may not be all that you could hope for, or need, at present, that could and should change in the future. What I learned in the writing of this article is that using App Engine for Java for scalable, data-intensive Java application development, today, is not for the timid or conservative. It's more like diving into the deep end of the pool: There's not a lifeguard in sight, and it's up to you whether your project sinks or swims.

Note that the example application in this article is based on the contact-management application developed in Part 2 of this article. You'll need to have that application built and runnable in order to proceed with the examples here.
Nuts and bolts, and leaky abstractions

Like the original Google App Engine, App Engine for Java relies on Google's internal infrastructure for the Big Three of scalable application development: distribution, replication, and load balancing. Because you're working with the Google infrastructure, most of this magic happens behind the scenes, accessible to you via App engine for Java's standards-based APIs. The datastore interface is based on JDO and the JPA, which are themselves based on the open source DataNucleus project. AppEngine for Java also provides a low-level adapter API for dealing directly with the App Engine for Java datastore, which is based on Google's BigTable implementation (see Part 1 for more about BigTable).

App Engine for Java data persistence isn't quite as straightforward as persistence in pure-Google App Engine, however. The JDO and JPA interfaces present some leaky abstractions due to the fact that BigTable is not a relational database. For example, in App Engine for Java, you can't do queries that do joins. You can set up relationships in JPA and JDO, but they can only be used for persisting relationships. And when you persist objects, they can only be persisted in the same atomic transaction if they are in the same entity group. By convention, relationships that are owned are in the same entity group as the parent. Conversely, un-owned relationships are in separate entity groups.

Rethinking data normalization

Working with App Engine's scalable datastore requires you to rethink your indoctrination to the benefits of normalized data. Of course, if you have worked long enough in the real world, you probably have sacrificed normalization on the altar of performance once or twice already. The difference is, when dealing with the App Engine datastore, you must de-normalize early and often. De-normalization is no longer a dirty word; instead, it is a design tool that you will apply in many aspects of your App Engine for Java applications.

The main drawback to App Engine for Java's leaky persistence comes up when you try to port an application written for an RDBMS to App Engine for Java. The App Engine for Java datastore is not a drop-in replacement for a relational database, so what you do with App Engine for Java may not be easy to translate to an RDBMS port. Taking an existing schema and porting it to the datastore is an even less likely scenario. If you do decide to port a legacy Java enterprise application to App engine, be advised to proceed with caution, and back it up with analysis. Google App Engine is a platform for applications designed for it specifically. Google App Engine for Java's support for JDO and JPA enables these apps to be ported back to more traditional, albeit un-normalized, enterprise applications.

The trouble with relationships

Another downside of App Engine for Java in its current preview release is its handling of relationships. In order to create relationships, you currently have to use extensions specific to App Engine for Java to JDO. Given that keys are generated based on artifacts of BigTable — that is, the "primary key" has the parent-object key encoded into all of its children — you will have to manage your data in a non-relational database. Another limitation is persisting data. Complexities arise if you use the non-standard AppEngine for Java Key class. First, how do you use the non-
standard key when porting your model to an RDBMS? Second, the Key class can't be translated by the GWT engine, so any model object that uses this class cannot be used as part of your GWT application.

Of course, at the time of this writing, Google App Engine for Java is very much in preview mode. It is not considered ready for prime-time. This becomes abundantly clear when studying the documentation of relationships in JDO, which is sparse and contains incomplete examples.

The App Engine for Java development kit ships with a series of example programs. Many of these examples use JDO, none use JPA. Not one of the examples (including the one called jdoexamples) has a single example of even a simple relationship. Instead, all of the examples use just one object to store data in the datastore. The Google App Engine for Java discussion group is flooded with questions about how to get simple relationships to work, with few answers. Some developers have apparently been able to get it working, but not without elbow grease and some complications.

The bottom line on relationships in App Engine for Java is that you need to manage them without much support from JDO or JPA. Google’s BigTable is a proven technology for producing scalable applications, however, and you can build on top of that. Building on top of BigTable frees you from dealing with an API facade that is less than fully baked. On the other hand, you will be dealing with a lower level API.

**Java Data Objects in AppEngine for Java**

While it may not make sense to port a traditional Java application into App Engine for Java, and even given the challenge of relationships, there still are persistence scenarios where it could make sense to use the platform. I'll conclude this article with a working example that should give you a taste of how App Engine for Java persistence works. We'll start with the contact-management application built in Part 2, this time walking through the procedure of adding support for persisting Contact objects with the App Engine for Java datastore facilities.

In the previous article, you created a simple GWT GUI to do CRUD operations on Contact objects. You defined the simple interface shown in Listing 1:

**Listing 1. The simple ContactDAO interface**

```java
package gaej.example.contact.server;
import java.util.List;
import gaej.example.contact.client.Contact;
public interface ContactDAO {
    void addContact(Contact contact);
    void removeContact(Contact contact);
    void updateContact(Contact contact);
    List<Contact> listContacts();
}
```

Next, you created a mock version that talked to data in an in-memory collection, as shown in Listing 2:
Listing 2. ContactDAOMock mocking the DAO

```java
package gaej.example.contact.server;
import gaej.example.contact.client.Contact;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class ContactDAOMock implements ContactDAO {

    Map<String, Contact> map = new LinkedHashMap<String, Contact>(){
        {map.put("rhightower@mammatus.com", new Contact("Rick Hightower", "rhightower@mammatus.com", "520-555-1212"));
         map.put("scott@mammatus.com", new Contact("Scott Fauerbach", "scott@mammatus.com", "520-555-1213"));
         map.put("bob@mammatus.com", new Contact("Bob Dean", "bob@mammatus.com", "520-555-1214"));
    }

    public void addContact(Contact contact) {
        String email = contact.getEmail();
        map.put(email, contact);
    }

    public List<Contact> listContacts() {
        return Collections.unmodifiableList(new ArrayList<Contact>(map.values()));
    }

    public void removeContact(Contact contact) {
        map.remove(contact.getEmail());
    }

    public void updateContact(Contact contact) {
        map.put(contact.getEmail(), contact);
    }
}
```

Now let's see what happens when you replace the mock implementation with a version of the application that interacts with the Google App Engine datastore. For this example, you'll use JDO to persist the Contact class. An application written using the Google Eclipse Plugin already has all the libraries it needs to use JDO. It also includes a jdoconfig.xml file, so once you annotate the Contact class, you'll be ready to start using JDO.

Listing 3 shows the ContactDAO interface extended to use the JDO API to persist, query, update, and delete objects:

Listing 3. ContactDAO with JDO

```java
package gaej.example.contact.server;
import gaej.example.contact.client.Contact;
import java.util.List;
import javax.jdo.JDOHelper;
```
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

public class ContactJdoDAO implements ContactDAO {
    private static final PersistenceManagerFactory pmfInstance = JDOHelper
            .getPersistenceManagerFactory("transactions-optional");

    public static PersistenceManagerFactory getPersistenceManagerFactory() {
        return pmfInstance;
    }

    public void addContact(Contact contact) {
        PersistenceManager pm = getPersistenceManagerFactory().
            getPersistenceManager();
        try {
            pm.makePersistent(contact);
        } finally {
            pm.close();
        }
    }

    @SuppressWarnings("unchecked")
    public List<Contact> listContacts() {
        PersistenceManager pm = getPersistenceManagerFactory().
            getPersistenceManager();
        String query = "select from " + Contact.class.getName();
        return (List<Contact>) pm.newQuery(query).execute();
    }

    public void removeContact(Contact contact) {
        PersistenceManager pm = getPersistenceManagerFactory().
            getPersistenceManager();
        try {
            pm.currentTransaction().begin();
            // We don't have a reference to the selected Product.
            // So we have to look it up first,
            contact = pm.getObjectById(Contact.class, contact.getId());
            pm.deletePersistent(contact);
            pm.currentTransaction().commit();
        } catch (Exception ex) {
            pm.currentTransaction().rollback();
            throw new RuntimeException(ex);
        } finally {
            pm.close();
        }
    }

    public void updateContact(Contact contact) {
        PersistenceManager pm = getPersistenceManagerFactory().
            getPersistenceManager();
        String name = contact.getName();
        String phone = contact.getPhone();
        String email = contact.getEmail();
        try {
            pm.currentTransaction().begin();
            // We don't have a reference to the selected Product.
            // So we have to look it up first,
            contact = pm.getObjectById(Contact.class, contact.getId());
            contact.setName(name);
            contact.setPhone(phone);
            contact.setEmail(email);
            pm.makePersistent(contact);
            pm.currentTransaction().commit();
        } catch (Exception ex) {
            pm.currentTransaction().rollback();
            throw new RuntimeException(ex);
        } finally {
            pm.close();
        }
    }
}
Method by method

Now let’s consider what’s happening with each of the methods in Listing 3. You’ll find that while the method names may be new, their action is for the most part familiar.

First, in order to access the PersistenceManager, you created a static PersistenceManagerFactory. If you’ve worked with JPA before, PersistenceManager is analogous to an EntityManager in JPA. If you’ve worked with Hibernate, PersistenceManager is similar to a Hibernate Session. Essentially, PersistenceManager is the main interface to the JDO persistence system. It represents the session to the database. The method getPersistenceManagerFactory() returns the statically initialized PersistenceManagerFactory, as shown in Listing 4:

### Listing 4. `getPersistenceManagerFactory()` returns PersistenceManagerFactory

```
private static final PersistenceManagerFactory pmfInstance = JDOHelper
    .getPersistenceManagerFactory("transactions-optional");

public static PersistenceManagerFactory getPersistenceManagerFactory() {
    return pmfInstance;
}
```

The addContact() method adds a new contact to the datastore. In order to do this, it needs to create an instance of a PersistenceManager and then call the makePersistence() method of the PersistenceManager. The makePersistence() method takes the transient Contact object (which the user will have filled out in the GWT GUI) and makes it a persistent object. All of this is shown in Listing 5:

### Listing 5. addContact()

```
public void addContact(Contact contact) {
    PersistenceManager pm = getPersistenceManagerFactory()
        .getPersistenceManager();
    try {
        pm.makePersistent(contact);
    } finally {
        pm.close();
    }
}
```

Notice in Listing 5 how the persistenceManager is enclosed in a finally block. This ensures it cleans up the resources associated with the persistenceManager.

The listContact() method, shown in Listing 6, creates a query object from the persistenceManager that it looks up. It invokes the execute() method, which returns the list of Contacts from the datastore.
Listing 6. listContact()

```java
@SuppressWarnings("unchecked")
public List<Contact> listContacts() {
    PersistenceManager pm = getPersistenceManagerFactory().
    .getPersistenceManager();
    String query = "select from " + Contact.class.getName();
    return (List<Contact>) pm.newQuery(query).execute();
}
```

The `removeContact()` method looks up the contact by ID before it removes it from the dataStore, as shown in Listing 7. It has to do this rather than just deleting the contact directly because the contact coming from the GWT GUI knows nothing about JDO. You have to get a contact that is associated with a PersistenceManager cache before you can delete it.

Listing 7. removeContact()

```java
public void removeContact(Contact contact) {
    PersistenceManager pm = getPersistenceManagerFactory().
    .getPersistenceManager();
    try {
        pm.currentTransaction().begin();
        // We don't have a reference to the selected Product.
        // So we have to look it up first,
        contact = pm.getObjectById(Contact.class, contact.getId());
        pm.deletePersistent(contact);
        pm.currentTransaction().commit();
    } catch (Exception ex) {
        pm.currentTransaction().rollback();
        throw new RuntimeException(ex);
    } finally {
        pm.close();
    }
}
```

The `updateContact()` method, shown in Listing 8, is similar to the `removeContact()` method, in that it looks up the contact. The `updateContact()` method then copies the properties from the contact. These properties are passed as an argument to the contact, which was looked up with the persistence manager. Objects that are looked up are checked for changes by the PersistenceManager. If an object has changed, the changes are flushed to the database by the PersistenceManager when a transaction commits.

Listing 8. updateContact()

```java
public void updateContact(Contact contact) {
    PersistenceManager pm = getPersistenceManagerFactory().
    .getPersistenceManager();
    try {
        pm.currentTransaction().begin();
        // We don't have a reference to the selected Product.
        // So we have to look it up first,
        contact = pm.getObjectById(Contact.class, contact.getId());
        contact.setName(name);
        contact.setPhone(phone);
        contact.setEmail(email);
        pm.currentTransaction().commit();
    } catch (Exception ex) {
        pm.currentTransaction().rollback();
        throw new RuntimeException(ex);
    } finally {
        pm.close();
    }
}
```
Annotation for object persistence

In order for a Contact to be persistable, you must identify it as a persistence-capable object with the @PersistenceCapable annotation. You then need to annotate all of its persistable fields, as shown in Listing 9:

Listing 9. Contact is persistable

```java
package gaej.example.contact.client;

import java.io.Serializable;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.IdentityType;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Contact implements Serializable {
    private static final long serialVersionUID = 1L;
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Long id;

    @Persistent
    private String name;

    @Persistent
    private String email;

    @Persistent
    private String phone;

    public Contact() {
    }

    public Contact(String name, String email, String phone) {
        super();
        this.name = name;
        this.email = email;
        this.phone = phone;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
    }
}
```
return name;
}

public void setName(String name) {
    this.name = name;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getPhone() {
    return phone;
}

public void setPhone(String phone) {
    this.phone = phone;
}

Due to the wonders of object-oriented programming and the principles of design by interface, you can simply replace your original ContactDAOMock with the new ContactJdoDAO. The GWT GUI will then work with JDO without changes.

Finally, what does change with this swap is the way that the DAO is instantiated in the service, as shown in Listing 10:

**Listing 10. RemoteServiceServlet**

```java
public class ContactServiceImpl extends RemoteServiceServlet implements ContactService {
    private static final long serialVersionUID = 1L;
    //private ContactDAO contactDAO = new ContactDAOMock();
    private ContactDAO contactDAO = new ContactJdoDAO();
    ...
}
```

**In conclusion**

In this three-part article, I've introduced Google App Engine for Java's current support for persistence, which is one of the cornerstones of scalable application delivery. The overall finding is disappointing, although it is important to keep in mind that this is an evolving platform. Applications that are written for the App Engine for Java preview release are tied to App Engine's persistence infrastructure, even though they might use JDO or JPA. App Engine for Java in its preview version also offers little documentation of its persistence framework, and the examples that App Engine for Java ships with make it nearly impossible to get even simple relationships working.

Even if the JDO and JPA implementations were fully cooked, it is currently unlikely that you could write a App Engine for Java application and port it easily to an RDBMS-based enterprise application. At the least, you would have to do some hefty coding to make the port work.

My hope for persistence is that it will mature with time. If you really need to work with App Engine for Java now, you will probably want to bypass the Java APIs and write directly to the low-
level Datastore API. Working with the App Engine for Java platform is possible, but if you are accustomed to working with JPA and/or JDO, you will encounter a learning curve, due to both the leaky abstraction described at the beginning of this article and features that either don't yet work well or aren't currently well documented.
Related topics

- "Google App Engine for Java, Part 1: Rev it up!" (Rick Hightower, developerWorks, August 2009): Get started with the Google Plugin for Eclipse and learn how to build simple applications quickly that scale.
- "Google App Engine for Java, Part 2: Building the killer app" (Rick Hightower, developerWorks, August 2009): Whet your appetite for what's possible with the App Engine, with this short tutorial in building a custom GWT GUI for a contact-management application. A prerequisite to complete the exercise in this article.
- "Connecting to the cloud, Part 1: Leverage the cloud in applications" (Mark O'Neill, developerWorks, April 2009): A three-part overview of cloud computing platforms from the major vendors: Amazon, Google, Microsoft®, and SalesForce.com.
- **JDO Tutorial, the Apache DB Project**: Learn how to use the `ObjectRelationalBridge` and the JDO API in a simple application scenario.
- **The DataNucleus Project**: An open source project that provides a series of software products around data management using the Java programming language; formerly Java Persistent Objects (JPOX).
- **Google Plugin for Eclipse**: Download the plugins to get started.
- **Google App Engine account**: You'll need one to deploy your killer app using the Google App Engine infrastructure.
- **Google App Engine homepage**: Learn more about the platform.

© Copyright IBM Corporation 2009
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)