Using the Google App Engine with Java

Michael Parker
michael.g.parker@gmail.com

App Engine Introduction

- Upload your web app to sandbox, and it's ready to go
  - The good: little maintenance, scalable transactional storage, secure and reliable environment, standard APIs used
  - The bad/unfamiliar: not a relational DB, sandboxed filesystem and sockets, no long-running responses
- Free quota: 500MB of storage, and CPU and bandwidth for 5M pageviews per month

Services

Service	Java Standard	Google Infrastructure
Authentication	Servlet API	Google Accounts
Datastore	JPA, JDO	Bigtable
Caching	javax.cache	memcacheg
E-mail	javax.mail	Gmail gateway
URLFetch	URLConnection	Caching HTTP Proxy

- Other services:
  - Java servlet 2.5 implementation, image manipulation, asynchronous task scheduling

Development

- Apache Ant component to simplify common App Engine tasks
- Google Plugin for Eclipse
- Local development server simulates the sandbox restrictions, datastore, and services
  - LRU memcache
  - Disk-backed datastore
  - Jakarta Commons HttpClient-backed URL Fetch

Sandboxing

- Can read all application files uploaded with the app; for read-write, use the datastore.
- No “direct” network access; use the URL fetch service for HTTP/HTTPS access
- No spawning new threads or processes; must use cron service
- Servlet requests can take up to 30s to respond before a throwing DeadlineExceededException

Datastore with JDO

- JDO (JSR 243) defines annotations for Java objects, retrieving objects with queries, and interacting with a database using transactions
- Post-compilation "enhancement" step on compiled classes associates them with the JDO implementation
- The PersistenceManager is the interface to the underlying JDO implementation
- Datastore implementation is scalable with an emphasis on reads and queries

Datastore Entities

- A entity has one or more properties, which are ints, floats, strings, dates, blobs, or references to other entites
- Each entity has a key; entities are fetched using their corresponding key, or by a query that matches its properties.
- Entities are schemaless; must enforce at the application level

Annotating an Entity with JDO

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Employee {
   @PrimaryKey
   @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
   private Long id;

   @Persistent
   private String firstName;

   @Persistent
   private String lastName;

   @Persistent
   private Date hireDate;

   Public Employee(String firstName, String lastname,
      Date hireDate) { … }

   /* accessors and other methods here */
}
```

Entity Keys

- Unique and identified by the @PrimaryKey annotation.
- Keys are a kind (class name) and:
  - A long automatically generated by the datastore, e.g. the unique message ID for an e-mail
  - A string specified by the client, e.g. the username belonging to an account

Creating Keys

- A Key instance combines the long or string fields with key representing the entity group ancestors, if any

```
Key keyFromString = KeyFactory.createKey(
      Employee.class.getSimpleName(), "Alfred.Smith@example.com");

Key keyFromLong = KeyFactory.createKey(
      Employee.class.getSimpleName(), 52234);

Key keyWithParent = new KeyFactory
      .Builder(Employee.class.getSimpleName(), 52234)
      .addChild(ExpenseReport.class.getSimpleName(), "A23Z79")
      .getKey();
```

Atomic Storage Operations

```
PersistenceManager pm = pmfInstance.getPersistenceManager();

Employee e = new Employee("Alfred", "Smith", new Date());
try {
    // Create
    pm.makePersistent(e);

    // Update
    Key key = KeyFactory.createKey(
            Employee.class.getSimpleName(),
            "Alfred.Smith@example.com");
    Employee copy = pm.getObjectById(Employee.class, key);

    // Delete
    pm.deletePersistent(copy);
} finally {
    pm.close();
}
```

Queries

- A query specifies
  - An entity kind
  - Zero or more conditions based on their property values
  - Zero or more sort orders
- Once executed, can return all entities meeting these criteria in the given sort order, or just their keys
- JDO has its own query language, like SQL, with two different calling styles

JDOQL Calling Styles

- String style:

```
Query query = pm.newQuery("select from Employee " +
      "where lastName == lastNameParam " +
      "order by hireDate desc " +
      "parameters String lastNameParam")
List<Employee> results = (List<Employee>) query.execute("Smith");
```

- Method style:

```
Query query = pm.newQuery(Employee.class);          // select from
query.setFilter("lastName == lastNameParam");       // where
query.setOrdering("hireDate desc");                 // order by
query.declareParameters("String lastNameParam");    // parameters
List<Employee> results = (List<Employee>) query.execute("Smith");
```

Query Caveats

- Filters have a field name, an operator, and a value
  - The value must be provided by the app
  - The operator must be in < <= == >= >
  - Only logical and is supported for multiple filters
  - Cannot test inequality on multiple properties
- A query can specify a range of results to be returned to the application.
  - Datastore must retrieve and discard all results before to the starting offset

Indexes

- An application has an index for each combination of kind, filter property and operator, and sort order used in a query.
- Given a query, the datastore identifies the index to use
  - all results for every possible query that uses an index are in consecutive rows in the table
- An index will sort entities first by value type, then by an order appropriate to the type.
  - Watch out! 38 (int) < 37.5 (float)

Custom Indexes

- In production, a query with no suitable index will fail, but the development web server can create the configuration for an index and succeed
  - Indexes specified in datastore-indexes.xml
- Must specify an index to be built for queries like:
  - queries with multiple sort orders
  - queries with a sort order on keys in descending order

Custom Indexes Code

- The XML configuration:

```
<?xml version="1.0" encoding="utf-8"?>
<datastore-indexes
  xmlns="http://appengine.google.com/ns/datastore-indexes/1.0"
  autoGenerate="true">
    <datastore-index kind="Person" ancestor="false">
       <property name="lastName" direction="asc" />
       <property name="height" direction="desc" />
    </datastore-index>
</datastore-indexes>
```

supports:

```
select from Person where lastName = 'Smith'
                    && height < 72
                  order by height desc
```

Exploding Indexes

- A property value for an entity is stored in every custom index that refers to the property
  - The more indexes that refer to a property, the longer it takes to update a property
- For properties with multiple values, an index has a row for every permutation of values for every property
- To keep updates quick, datastore limits the number of index entries an entity can have
  - Insertion or update will fail with an exception

Exploding Indexes Example

- Custom index:

```
<?xml version="1.0" encoding="utf-8"?>
<datastore-indexes>
   <datastore-index kind="MyModel">
      <property name="x" direction="asc" />
      <property name="y" direction="asc" />
   </datastore-index>
</datastore-indexes>
```

Built-in on x:

one, two
two, one

Built-in on y:

three, four
four, three

- Adding an entity:

```
MyModel m = new MyModel();
m.setX(Arrays.asList("one", "two"));
m.setY(Arrays.asList("three", "four"));
pm.makePersistent(m);
```

Custom index:

one, two	three, four
one, two	four, three
two, one	three, four
two, one	four, three

Relationships

- Relationship dimensions:
  - Owned versus unowned
  - One-to-one versus one-to-many
  - Unidirectional and bidirectional
- Implementation of the JDO can model owned one-to-one and owned one-to-many relationships, both unidirectional and bidirectional
  - Unowned is possible with some manual bookkeeping, allows many-to-many

Owned, one-to-one

- Have a parent (the owner) and a child
  - Follows from encapsulation in code
  - Child key uses the parent key as its entity group parent
- When the parent is retrieved, the child is retrieved
- In unidirectional case, child has Key for parent
- In bidirectional case, child has reference to parent
  - When child is retrieved, parent is retrieved

One-to-one unidirectional code

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class ContactInfo /* the child */ {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

   // ...
}

@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Employee /* the parent */ {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Long id;

    @Persistent
    private ContactInfo contactInfo;

    // …
}
```

One-to-one bidirectional code

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class ContactInfo /* the child */ {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent(mappedBy = "contactInfo")
    private Employee employee;

  // ...
}
```

- Note that the Key member is still present
- The argument to mappedBy must be the name of the child in the parent class

One-to-many bidirectional code

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class ContactInfo /* the child */ {
    // ...
    @Persistent
    private Employee employee;
    // ...
}

@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Employee /* the parent */ {
    // …
    @Persistent(mappedBy = “employee”)
    private List<ContactInfo> contactInfoSets;
    // …
}
```

- Note that mappedBy is on the parent class, its argument is its name in the child class

Owned collections

- Can use any Set, List, or built-in collection implementation for owned one-to-many
- Order is preserved by storing a position property for every element
  - If an element is added or deleted, positions of subsequent elements must be updated
- If you do not need to preserve arbitrary order, use the @Order annotation:

```
@Persistent
@Order(extensions = @Extension(vendorName="datanucleus",
   key="list-ordering", value="state asc, city asc"))
private List<ContactInfo> contactInfoSets = new List<ContactInfo>();
```

Unowned relationships

- Use Key instances instead of instances or a collection of instances
- Easy to model any relationship, but
  - No referential integrity is enforced
  - In some cases, entities on different sides of the relationship belong to different entity groups, disallowing atomic updates

Unowned many-to-many code

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Person {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Long id;

    @Persistent
    private Set<Key> favoriteFoods;
}

@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class Food {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Long id;

    @Persistent
    private Set<Key> foodFans;
}
```

Relationships and Transactions

```
Employee e = new Employee();
ContactInfo ci = new ContactInfo();
e.setContactInfo(ci);
pm.makePersistent(e);
```

- Without a transaction, entities are created in separate atomic actions, not a single one:

```
Transaction tx = null;
try {
    tx = pm.currentTransaction();
    tx.begin();
    pm.makePersistent(e);
    tx.commit();
} finally {
    if (tx.isActive()) {
        tx.rollback();
    }
}
```

Low Level Datastore API

- There is a lower-level API if you don't like the abstraction that JDO provides you
  - This is the API that the App Engine JDO implementation uses
- Data store operations:
  - get for set of keys with optional transaction
  - put for set of values with optional transaction
  - delete for set of keys with optional transaction
  - query preparation and execution

Entity Groups

- The fundamental data unit in a transaction is the entity group; a single transaction can only manipulate data in one entity group.
- Each entity group is a hierarchy:
  - An entity without a parent is a root entity.
  - An entity that is a parent for another entity can also have a parent.
  - Every entity with a given root entity as an ancestor is in the same entity group.
- All entities in a group are stored in the same datastore node.

Creating a Hierarchy

- Creating a hierarchical data model is very different from using SQL
- For example, given a online photo album, can define the user as the root
  - children can be preferences and photo albums
  - children of albums can be images, which can be further broken down into EXIF data and comments, etc.

Hierarchies with JDO

```
@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class AccountInfo {
   @PrimaryKey
   @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
   private Key key;

   public void setKey(Key key) {
      this.key = key;
   }
}

public static void createAccount(String customerId, String accountId) {
   KeyFactory.Builder keyBuilder = new KeyFactory.Builder(
         Customer.class.getSimpleName(), customerId);
   keyBuilder.addChild(AccountInfo.class.getSimpleName(),
         accountId);
   Key accountKey = keyBuilder.getKey();
   return new AccountInfo(customerId, accountId);
}
```

Transactions

- When a transaction commits, all writes succeed, or else the transaction fails and must be retried.
- A transaction uses optimistic concurrency control:
  - When creating the transaction, get the time the entity group was last updated
  - On every read within the group, succeed if the entity group time is unchanged
  - On committing, or writing, succeed if the entity group time is unchanged

Transaction Help

- With optimistic concurrency, can need to try a transaction several times; JDO throws a JDODataStore exception and gives up
  - Consider bundling the transaction logic into a Runnable or Callable, and have a helper method
- Make them happen quickly
  - Prepare keys and data outside the transaction

Transactions with JDO

```
for (int i = 0; i < NUM_RETRIES; i++) {
    pm.currentTransaction().begin();

    ClubMembers members = pm.getObjectById(
            ClubMembers.class, "k12345");
    members.incrementCounterBy(1);

    try {
        pm.currentTransaction().commit();
        break;

    } catch (JDOCanRetryException ex) {
        if (i == (NUM_RETRIES - 1)) {
            throw ex;
        }
    }
}
```

Memcache

- Implementation of JCache (JSR 107) atop of memcache
- Use when you would a traditional cache:
  - The data is popular or query is expensive
  - Returned data can be potentially stale
  - If the cached data is unavailable, the application performs fine
- Entries evicted in LRU order when low on memory, or an expiration time can be provided
- Like the datastore, has a low-level API

Memcache code

- Behaves like java.util.Map:

```
String key = "key";
byte[] value = "value".getBytes(Charset.forName(“UTF-8”));

// Put the value into the cache.
cache.put(key, value);
// Get the value from the cache.
value = (byte[]) cache.get(key);
```

- Has other familiar methods, like putAll, containsKey, size, isEmpty, remove, and clear
- Can set the policy when a value exists
  - Has “only replace” as well as “only add”

URL Fetch

- Synchronous HTTP or HTTPS retrieval allowing GET, POST, PUT, HEAD, and DELETE through a HTTP/1.1-compliant proxy
- Can set HTTP headers on outgoing requests
  - Some exceptions, e.g. Host and Referer
- Use Google Secure Data Connector to access intranet URLs
  - Restricts to users signed in using an Apps account for your domain
- Like the memcache, has a low-level API

URL Fetch Code

- URL.openStream() transparently uses URL fetch:

```
URL url = new URL("http://www.example.com/atom.xml");
BufferedReader reader = new BufferedReader(
        new InputStreamReader(url.openStream()));
```

- As will URL.openConnection():

```
URL url = new URL("http://www.example.com/comment");
HttpURLConnection connection =
        (HttpURLConnection) url.openConnection();
```

- URLConnection is not persistent; buffers request until the client accesses the response, and once received, closes the connection.

Mail

- To not use the administrator's e-mail account as sender, create a new account and add it as the administrator for the application
- When mail service is called, message is enqueued and call returns immediately
  - Application receives no notification of whether delivery succeeded or failed
- Sender is the application developer or the address of the Google Accounts user
- Like URL fetch, has a low-level API

Google Accounts

- Authentication with Google Accounts is optional
  - Address from Apps domain, or gmail.com
  - Allows development of admin-only site parts
- Not SSO from other Google applications
- Datastore supports storing the User object as a special value type, but don't rely on them as stable user identifiers
  - If a user changes his or her e-mail address, new User is different than what is stored

Google Accounts Code

```
UserService userService = UserServiceFactory.getUserService();

String thisURL = request.getRequestURI();
if (request.getUserPrincipal() != null) {
    response.getWriter().println("<p>Hello, " +
                                 request.getUserPrincipal().getName() +
                                 "!  You can <a href=\"" +
                                 userService.createLogoutURL(thisURL) +
                                 "\">sign out</a>.</p>");
} else {
    response.getWriter().println("<p>Please <a href=\"" +
                                 userService.createLoginURL(thisURL) +
                                 "\">sign in</a>.</p>");
}
```

Deployment Descriptor

- The web.xml in the application's WAR file under the WEB-INF/ directory defines URL to servlet mappings, which URLs require auth, and other properties
  - Part of the servlet standard, many references
- App Engine supports automatic compilation and URL mapping for JSPs, and the JSP Standard Tag Library

Deployment Security

- A user role of * requires a Google Account:

```
<security-constraint>
    <web-resource-collection>
        <url-pattern>/profile/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>*</role-name>
    </auth-constraint>
</security-constraint>

<security-constraint>
    <web-resource-collection>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
```

Application Configuration

- An appengine-web.xml file specifies additional application properties
  - The application identifier
  - The version identifier of the latest code
  - Static files (publically served) and resource files (application private)
  - System properties and environment variables
  - Toggling of SSL and sessions

Scheduled Tasks

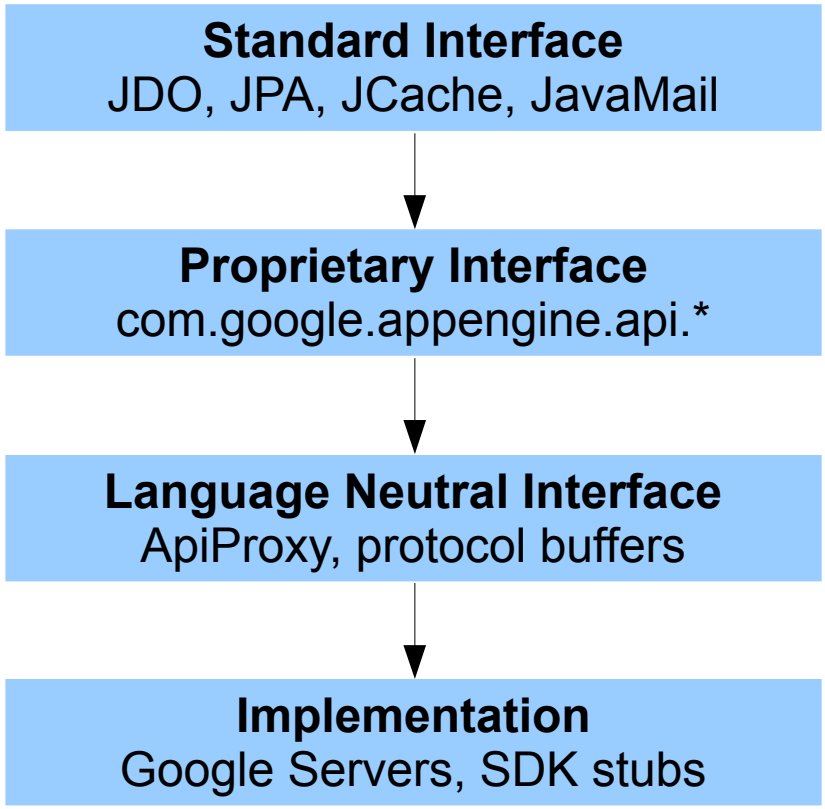
- No Executor service for task scheduling yet...
- The cron service invokes a URL at a specified time of day
  - Scheduled tasks can access admin-only URLs
  - Requests have the HTTP header X-AppEngine-Cron: true
- Time in a simple English-like format:
  - every 5 minutes
  - 2nd,third mon,wed,thu of march 17:00
  - every day 00:00

Scheduled Tasks cron.xml

- Timezone is UTC (i.e. GMT) by default:

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/recache</url>
    <description>Repopulate the cache every 2 minutes</description>
    <schedule>every 2 minutes</schedule>
  </cron>
  <cron>
    <url>/weeklyreport</url>
    <description>Mail out a weekly report</description>
    <schedule>every monday 08:30</schedule>
    <timezone>America/New_York</timezone>
  </cron>
</cronentries>
```

Service Implementation



- All calls go through ApiProxy, which in turn invokes a registered delegate

Profiling with ApiProxy

```
class ProfilingDelegate extends Delegate {
    Delegate parent;
    public ProfilingDelegate(Delegate parent) {
        this.parent = parent;
    }
    public byte[] makeSyncCall(Environment env, String pkg,
            String method, byte[] request) {
        long start = System.nanoTime();
        byte[] result = parent.makeSyncCall(env, pkg, method, request);
        log.log(INFO,
                pkg + “.” + method + “: “ + System.nanoTime() - start);
        return result;
    }
}

ApiProxy.setDelegate(new ProfilingDelegate(ApiProxy.getDelegate()));
```

Defining the Test Environment

- Implement ApiProxy.Environment to return information that appengine-web.xml returns:

```
class TestEnvironment implements ApiProxy.Environment {
    public String getAppId() {
        return "Unit Tests";
    }
    public String getVersionId() {
        return "1.0";
    }
    public String getAuthDomain() {
        return "gmail.com";
    }

    // ...
}
```

Creating a Test Harness

- Specify the local implementations of all services, so you do not use method stubs to a remote server:

```
public class LocalServiceTestCase extends TestCase {
   @Override
   public void setUp() throws Exception {
      super.setUp();
      ApiProxy.setEnvironmentForCurrentThread(
            new TestEnvironment());
      ApiProxy.setDelegate(new ApiProxyLocalImpl(new File(".")));
   }

   // ...
}
```

Using the Test Harness

- Cast services to their local implementations:

```
public void testEmailGetsSent() {
    ApiProxyLocalImpl proxy =
        (ApiProxyLocalImpl) ApiProxy.getDelegate();
    LocalMailService mailService =
        (LocalMailService) proxy.getService("mail");
    mailService.clearSentMessages();

    Bug b = new Bug();
    b.setSeverity(Severity.LOW);
    b.setText("NullPointerException when updating phone number.");
    b.setOwner("max");
    new BugDAO().createBug(b);

    assertEquals(1, mailService.getSentMessages().size());
    // ... test the content and recipient of the email
}
```