Google App Engine Java and GWT Application Development

Daniel Guermeur

Amy Unruh

Community Experience Distilled
Google App Engine Java and GWT Application Development
Build powerful, scalable, and interactive web applications in the cloud
Daniel Guermeur
Amy Unruh
PACKT

Chapter No.9

"Robustness and Scalability: Transactions, Memcache, and Datastore Design"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.9 "Robustness and Scalability: Transactions, Memcache, and Datastore Design"

A synopsis of the book's content

Information on where to buy this book

About the Authors

Daniel Guermeur is the founder and CEO of Metadot Corporation. He holds a Diplome d'Ingenieur of Informatique from University of Technology of Compiegne (France) as well as a Master in Photonics from Ecole Nationale Superieure de Physique of Strasbourg (France). Before starting Metadot in 2000, he worked for oil services companies including giant Schlumberger Ltd where he was in charge of improving the worldwide IT infrastructure.

He has been developing large scale database-backed web applications since the very beginning of the democratization of the Internet in 1995, including an open source software content management system Metadot Portal Server, Mojo Helpdesk, a webbased customer support application and Montastic, a popular website monitor service.

Thank you to my daughter Alexandra Guermeur and Cheryl Ridall for their love and continuous support while writing this book. This meant a lot to me. Cheryl, I miss you.

Amy Unruh currently does technology training and course development, with a focus on web technologies. Previously, she has been a Research Fellow at the University of Melbourne, where she taught web technologies and researched robust distributed agent systems. She has worked at several startups, building web applications using a variety of languages; served as adjunct faculty at the University of Texas; and was a member of the Distributed Agents Program at MCC. She received her Ph.D. in CS/AI from Stanford University, in the area of AI planning, and has a BS degree in CS from UCSB. She has numerous publications, and has co-edited a book on Safety and Security in Multiagent Systems.

Thanks to Moon, and to Wesley and Eleanor Unruh, without whose love and support this book would not have been possible. And thanks also to Daniel, for getting us started!

Google App Engine Java and GWT Application Development

This book is designed to give developers the tools they need to build their own Google App Engine (GAE) with Google Web Toolkit (GWT) applications, with a particular focus on some of the technologies useful for building social-media-oriented applications. The book is centered on a GAE + GWT Java application called *Connectr*, which is developed throughout the chapters and demonstrates, by example, the use of the technologies described in the book. The application includes social-media information gathering and aggregation activities and incorporates the use of many App Engine services and APIs, as well as GWT design patterns and widget examples.

Several stages of the *Connectr* application are used throughout the book as features are added to the app. Code is included with the book for all application stages, and each chapter indicates the stage used.

What This Book Covers

Chapter 1, *Introduction,* introduces the approaches and technology covered in the book, and discusses what lies ahead.

Chapter 2, *Using Eclipse and the Google Plugin*, describes the basics of setting up a project using the Eclipse IDE and Google's GWT/GAE plugin. Topics include defining, compiling and running an Eclipse GWT/GAE project, and using the GWT developer browser plugin with the interactive debugger. The chapter also covers how to set up an App Engine account and create applications, and how to deploy an app to App Engine and access its Admin Console.

Chapter 3, Building The Connectr User Interface with GWT, focuses on GWT, and building the fi rst iteration of the *Connectr* application's frontend. The chapter looks at how to specify widgets, with a focus on declarative specification using GWT's UIBinder and using the GWT RPC API for server-side communication.

Chapter 4, *Persisting Data: The App Engine Datastore*, covers Datastore basics. In the process, the first iteration of *Connectr's* server-side functionality is built. The chapter looks at how the Datastore works, and the implications of its design for your data models and code development. It covers how to use Java Data Objects (JDO) as an interface to the Datastore and how to persist and retrieve Datastore entities.

Chapter 5, *JDO Object Relationships and Queries*, builds on the topics of *Chapter 4*. It describes how to build and manage JDO objects that have relationships to each other, such as one-to-many and one-to-one parent-child relationships. It also covers how to query the Datastore, and the important role that Datastore indexes play in this process.

Chapter 6, *Implementing MVP, an Event Bus and Other GWT Patterns*, builds on the client-side code of *Chapter 3*, and shows how to make the frontend code modular and extensible. It accomplishes this via use of the MVP (Model-View-Presenter) and Event Bus design patterns, history/bookmark management, and an RPC abstraction, which supports call retries and progress indicators.

Chapter 7, *Background Processing and Feed Management*, centers on defi ning and running decoupled backend asynchronous tasks. In the process, the chapter introduces several App Engine services, including URLFetch and Task Queues, shows the use of Query Cursors to distribute Datastore-related processing across multiple Tasks, and introduces the use of Java Servlets and the incorporation of third-party libraries in a deployed application.

Chapter 8, *Authentication using Twitter and Facebook OAuth and Google Accounts*, adds authentication, login, and account functionality to *Connectr*, allowing it to support multiple users. The chapter demonstrates the use of both the Google Accounts API and the OAuth protocol for creating user accounts.

Chapter 9, *Robustness and Scalability: Transactions, Memcache, and Datastore Design*, delves into more advanced Datastore-related topics. The chapter investigates Datastore-related means of increasing the robustness, speed, and scalability of an App Engine app, including several ways to design data classes for scalability and to support efficient join-like queries. The chapter also introduces App Engine transactions and Transactional Tasks and the use of Memcache, App Engine's volatile-memory key-value store.

Chapter 10, *Pushing fresh content to clients with the Channel API*, covers the implementation of a message push system using the App Engine Channel API, used by *Connectr* to keep application data streams current. The chapter describes how to open back-end channels connected to client-side socket listeners, and presents a strategy for preventing the server from pushing messages to unattended web clients.

Chapter 11, *Managing and Backing Up Your App Engine Application*, focuses on useful App Engine deployment strategies, and admin and tuning tools. It includes ways to quickly upload configuration fi les without redeploying your entire application and describes how to do bulk uploads and downloads of application data. The chapter also discusses tools to analyze and tune your application's behavior, and the App Engine billing model.

Chapter 12, *Asynchronous Processing with Cron, Task Queue, and XMPP*, finishes building the server-side part of the *Connectr* app. The chapter introduces the use of App Engine Cron jobs, configuration of customized Task Queues, and App Engine's XMPP service and API, which supports push notifications. The chapter shows the benefits of proactive and asynchronous updating—the behind-the scenes work that keeps *Connectr's* data stream fresh—and looks at how App Engine apps can both send and receive XMPP messages.

Chapter 13, Conclusion, summarizes some of the approaches and technology covered in the book, and discusses what might lie ahead.

9

Robustness and Scalability: Transactions, Memcache, and Datastore Design

Chapter 4 and *Chapter 5* explored the basics of using the App Engine Datastore. In this chapter, we'll delve deeper to investigate Datastore-related ways to help increase the robustness, speed, and scalability of an App Engine app, and apply these techniques to our *Connectr* app.

First, in the *Data modeling and scalability* section we look at ways to structure and access your data objects to make your application faster and more scalable.

Then, the *Using transactions* section describes the Datastore **transactions**, what they do, and when and how to use them. Finally, *Using Memcache* will introduce App Engine's **Memcache** service, which provides a volatile-memory key-value store, and discuss the use of Memcache to speed up your app.

In this chapter, we will use for our examples the full version of the *Connectr* app, `ConnectrFinal`.

Data modeling and scalability

In deciding how to design your application's data models, there are a number of ways in which your approach can increase the app's scalability and responsiveness. In this section, we discuss several such approaches and how they are applied in the *Connectr* app. In particular, we describe how the Datastore access latency can sometimes be reduced; ways to split data models across entities to increase the efficiency of data object access and use; and how property lists can be used to support "join-like" behavior with Datastore entities.

Reducing latency—read consistency and Datastore access deadlines

By default, when an entity is updated in the Datastore, all subsequent reads of that entity will see the update at the same time; this is called **strong consistency**. To achieve it, each entity has a primary storage location, and with a strongly consistent read, the read waits for a machine at that location to become available. Strong consistency is the default in App Engine.

However, App Engine allows you to change this default and use **eventual consistency** for a given Datastore read. With eventual consistency, the query may access a copy of the data from a secondary location if the primary location is temporarily unavailable. Changes to data will propagate to the secondary locations fairly quickly, but it is possible that an "eventually consistent" read may access a secondary location before the changes have been incorporated. However, eventually consistent reads are faster on average, so they trade consistency for availability. In many contexts, for example, with web apps such as *Connectr* that display "activity stream" information, this is an acceptable tradeoff—completely up-to-date freshness of information is not required.

This touches on a complex and interesting field beyond the scope of this book. See `http://googleappengine.blogspot.com/2010/03/read-consistency-deadlines-more-control.html`, `http://googleappengine.blogspot.com/2009/09/migration-to-better-datastore.html`, and `http://code.google.com/events/io/2009/sessions/TransactionsAcrossDatacenters.html` for more background on this and related topics.

In *Connectr*, we will add the use of eventual consistency to some of our feed object reads; specifically, those for feed content updates. We are willing to take the small chance that a feed object is slightly out-of-date in order to have the advantage of quicker reads on these objects.

The following code shows how to set eventual read consistency for a query, using `server.servlets.FeedUpdateFriendServlet` as an example.

```
    Query q = pm.newQuery("select from " + FeedInfo.class.getName() +
      "where urlstring == :keys");
    //Use eventual read consistency for this query
    q.addExtension("datanucleus.appengine.datastoreReadConsistency",
      "EVENTUAL");
```

App Engine also allows you to change the default Datastore *access deadline*. By default, the Datastore will retry access automatically for up to about 30 seconds. You can set this deadline to a smaller amount of time. It can often be appropriate to set a shorter deadline if you are concerned with response latency, and are willing to use a cached version of the data for which you got the timeout, or are willing to do without it.

The following code shows how to set an access timeout interval (in milliseconds) for a given JDO query.

```
  Query q = pm.newQuery("…");
  // Set a Datastore access timeout
  q.setTimeoutMillis(10000);
```

Splitting big data models into multiple entities to make access more efficient

Often, the fields in a data model can be divided into two groups: main and/or summary information that you need often/first, and *details*—the data that you might not need or tend not to need immediately. If this is the case, then it can be productive to split the data model into multiple entities and set the *details* entity to be a child of the summary entity, for instance, by using JDO owned relationships. The child field will be fetched lazily, and so the child entity won't be pulled in from the Datastore unless needed.

In our app, the `Friend` model can be viewed like this: initially, only a certain amount of summary information about each `Friend` is sent over RPC to the app's frontend (the Friend's name). Only if there is a request to view details of or edit a particular `Friend`, is more information needed.

So, we can make retrieval more efficient by defining a parent *summary* entity, and a child *details* entity. We do this by keeping the "summary" information in `Friend`, and placing "details" in a `FriendDetails` object, which is set as a child of `Friend` via a JDO bidirectional, one-to-one owned relationship, as shown in Figure 1. We store the `Friend`'s e-mail address and its list of associated URLs in `FriendDetails`. We'll keep the name information in `Friend`. That way, when we construct the initial 'FriendSummaries' list displayed on application load, and send it over RPC, we only need to access the summary object.

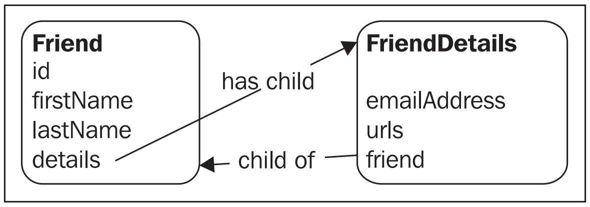

Figure 1: Splitting `Friend` data between a "main" `Friend` persistent class and a `FriendDetails` child class.

A `details` field of `Friend` points to the `FriendDetails` child, which we create when we create a `Friend`. In this way, the details will always be transparently available when we need them, but they will be lazily fetched—the `details` child object won't be initially retrieved from the database when we query `Friend`, and won't be fetched unless we need that information.

As you may have noticed, the `Friend` model is already set up in this manner—this is the rationale for that design.

Discussion

When splitting a data model like this, consider the queries your app will perform and how the design of the data objects will support those queries. For example, if your app often needs to query for `property1 == x` and `property2 == y`, and especially if both individual filters can produce large result sets, you are probably better off keeping both those properties on the same entity (for example, retaining both fields on the "main" entity, rather than moving one to a "details" entity).

For persistent classes (that is, "data classes") that you often access and update, it is also worth considering whether any of its fields do not require indexes. This would be the case if you never perform a query which includes that field. The fewer the indexed fields of a persistent class, the quicker are the writes of objects of that class.

Splitting a model by creating an "index" and a "data" entity

You can also consider splitting a model if you identify fields that you access only when performing queries, but don't require once you've actually retrieved the object. Often, this is the case with multi-valued properties. For example, in the *Connectr* app, this is the case with the `friendKeys` list of the `server.domain.FeedIndex` class (first encountered in *Chapter 7*). This multi-valued property is used to find relevant feed objects but is not used when displaying feed content information.

With App Engine, there is no way for a query to retrieve only the fields that you need (with the exception of *keys-only* queries, as introduced in *Chapter 5*), so the full object must always be pulled in. If the multi-valued property lists are long, this is inefficient.

To avoid this inefficiency, we can split up such a model into two parts, and put each one in a different entity—an index entity and a data entity. The **index entity** holds only the multi-valued properties (or other data) used only for querying, and the **data entity** holds the information that we actually want to use once we've identified the relevant objects. The trick to this new design is that the data entity key is defined to be the parent of the index entity key.

More specifically, when an entity is created, its key can be defined as a "child" of another entity's key, which becomes its parent. The child is then in the same **entity group** as the parent (we discuss entity groups further in the *Using transactions* section). Because such a child key is based on the path of its parent key, it is possible to derive the parent key given only the child key, using the `getParent()` method of `Key`, without requiring the child to be instantiated.

So with this design, we can first do a *keys-only* query on the index kind (which is faster than full object retrieval) to get a list of the keys of the relevant index entities. With that list, even though we've not actually retrieved the index objects themselves, we can derive the parent data entity keys from the index entity keys. We can then do a *batch fetch* with the list of relevant parent keys to grab all the data entities at once. This lets us retrieve the information we're interested in, without having to retrieve the properties that we do not need.

See Brett Slatkin's presentation, *Building scalable, complex apps on App Engine* (`http://code.google.com/events/io/2009/sessions/BuildingScalableComplexApps.html`) for more on this index/data design.

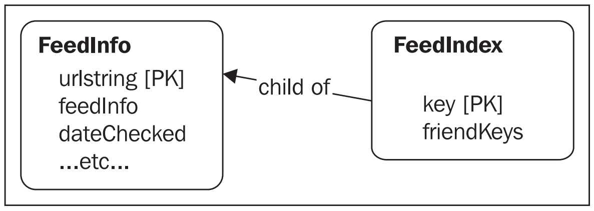

Figure 2: Splitting the feed model into an "index" part (`server.domain.FeedIndex`) and a "data" part (`server.domain.FeedInfo`)

Our feed model (which was introduced in *Chapter 7*) maps well to this design—we filter on the `FeedIndex.friendKeys` multi-valued property (which contains the list of keys of Friends that point to this feed) when we query for the feeds associated with a given `Friend`.

But, once we have retrieved those feeds, we don't need the `friendKeys` list further. So, we would like to avoid retrieving them along with the feed content. With our app's sample data, these property lists will not comprise a lot of data, but they would be likely to do so if the app was scaled up. For example, many users might have the same friends, or many different contacts might include the same company blog in their associated feeds.

So, we split up the feed model into an index part and a parent data part, as shown in Figure 2. The index class is `server.domain.FeedIndex`; it contains the `friendKeys` list for a feed. The data part, containing the actual feed content, is `server.domain.FeedInfo`. When a new `FeedIndex` object is created, its key will be constructed so that its corresponding `FeedInfo` object's key is its parent key. This construction must of course take place at object creation, as Datastore entity keys cannot be changed.

For a small-scale app, the payoff from this split model would perhaps not be worth it. But for the sake of example, let's assume that we expect our app to grow significantly.

The `FeedInfo` persistent class—the parent class—simply uses an app-assigned `String` primary key, `urlstring` (the feed URL string). The `server.domain.FeedIndex` constructor, shown in the code below, uses the key of its `FeedInfo` parent—the URL string—to construct its key. (The *Using transactions* section will describe the key-construction code of the figure in more detail). This places the two entities into the same **entity group** and allows the parent `FeedInfo` key to be derived from the `FeedIndex` entity's key.

```
@PersistenceCapable(identityType = IdentityType.APPLICATION,
   detachable="true")
public class FeedIndex implements Serializable {

  @PrimaryKey
  @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
  private Key key;
  …

  public FeedIndex(String fkey, String url) {
    this.friendKeys = new HashSet<String>();
    this.friendKeys.add(fkey);
    KeyFactory.Builder keyBuilder =
       new KeyFactory.Builder(FeedInfo.class.getSimpleName(), url);
    keyBuilder.addChild(FeedIndex.class.getSimpleName(), url);
    Key ckey = keyBuilder.getKey();
    this.key= ckey;
  }
```

The following code, from `server.servlets.FeedUpdateFriendServlet`, shows how this model is used to efficiently retrieve the `FeedInfo` objects associated with a given `Friend`. Given a `Friend` key, a query is performed for the *keys* of the `FeedIndex` entities that contain this `Friend` key in their `friendKeys` list. Because this is a *keys-only* query, it is much more efficient than returning the actual objects. Then, each `FeedIndex` key is used to derive the parent (`FeedInfo`) key. Using that list of parent keys, a *batch fetch* is performed to fetch the `FeedInfo` objects associated with the given `Friend`. We did this without needing to actually fetch the `FeedIndex` objects.

```
… imports…
@SuppressWarnings("serial")
public class FeedUpdateFriendServlet extends HttpServlet{

 private static Logger logger =
    Logger.getLogger(FeedUpdateFriendServlet.class.getName());
```

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)
   throws IOException {

  PersistenceManager pm = PMF.get().getPersistenceManager();

  Query q = null;
  try {
    String fkey = req.getParameter("fkey");
    if (fkey != null) {
      logger.info("in FeedUpdateFriendServlet, updating feeds for:"
         +fkey);
      // query for matching FeedIndex keys
      q = pm.newQuery("select key from "+FeedIndex.class.getName()+"
          where friendKeys == :id");
      List ids=(List)q.execute(fkey);
      if (ids.size()==0) {
        return;
      }
      // else, get the parent keys of the ids
      Key k = null;
      List<Key>parent list = new ArrayList<Key>();
      for (Object id : ids) {
        // cast to key
        k = (Key)id;
        parentlist.add(k.getParent());
      }
      // fetch the parents using the keys
      Query q2 = pm.newQuery("select from +FeedInfo.class.getName()+
         "where urlstring == :keys");
      // allow eventual consistency on read
      q2.addExtension(
       "datanucleus.appengine.datastoreReadConsistency",
       "EVENTUAL");
      List<FeedInfo>results =
        (List<FeedInfo>)q2.execute(parentlist);
      if(results.iterator().hasNext()){
         for(FeedInfo fi : results){
           fi.updateRequestedFeed(pm);
         }
      }
    }
  }
  catch (Exception e) {
     logger.warning(e.getMessage());
  }
```

```
     finally {
       if q!=null) {
         q.closeAll();
       }
       pm.close();
     }
    }
   }//end class
```

Use of property lists to support "join" behavior

Google App Engine does not support joins with the same generality as a relational database. However, property lists along with accompanying denormalization can often be used in GAE to support join-like functionality in a very efficient manner.

> At the time of writing, there is GAE work in progress to support simple joins. However, this functionality is not yet officially part of the SDK.

Consider the many-to-many relationship between `Friend` and feed information in our application. With a relational database, we might support this relationship by using three tables: one for Friend data, one for Feed data, and a "join table" (sometimes called a "cross-reference table"), named, say, `FeedFriend`, with two columns—one for the friend ID and one for the feed ID. The rows in the join table would indicate which feeds were associated with which friends.

In our hypothetical relational database, a query to find the feeds associated with a given `Friend fid` would look something like this:

```
select feed.feedname from Feed feed, FeedFriend ff
  where ff.friendid = 'fid' and ff.feedid = feed.id
```

If we wanted to find those feeds that *both* Friend 1 (`fid1`) and Friend 2 (`fid2`) had listed, the query would look something like this:

```
select feed.feedname from Feed feed, FeedFriend f1, FeedFriend f2
  where f1.friendid = 'fid1' and f1.feedid = feed.id
  and f2.friendid = 'fid2' and f2.feedid = feed.id
```

With Google App Engine, to support this type of query, we can denormalize the "join table" information and use Datastore multi-valued properties to hold the denormalized information. (Denormalization should not be considered a second-class citizen in GAE).

In *Connectr*, feed objects hold a list of the keys of the `Friends` that list that feed (`friendKeys`), and each `Friend` holds a list of the feed `URLs` associated with it. *Chapter 7* included a figure illustrating this many-to-many relationship.

So, with the first query above, the analogous JDQL query is:

```
select from FeedIndex where friendKeys == 'fid'
```

If we want to find those feeds that are listed by *both* Friend 1 and Friend 2, the JDQL query is:

```
select from FeedIndex where friendKeys == 'fid1' and
   friendKeys == 'fid2'
```

Our data model, and its use of multi-valued properties, has allowed these queries to be very straightforward and efficient in GAE.

Supporting the semantics of more complex joins

The semantics of more complex join queries can sometimes be supported in GAE with multiple synchronously-ordered multi-valued properties.

For example, suppose we decided to categorize the associated feeds `of` Friends by whether they were "Technical", "PR", "Personal", "Photography-related", and so on (and that we had some way of determining this categorization on a feed-by-feed basis). Then, suppose we wanted to find all the Friends whose feeds include "PR" feed(s), and to list those feed URLs for each `Friend`.

In a relational database, we might support this by adding a "Category" table to hold category names and IDs, and adding a category ID column to the Feed table. Then, the query might look like this:

```
select f.lastName, feed.feedname from Friend f, Category c,
  Feed feed, FeedFriend ff
  where c.id = 'PR' and feed.cat_id = c.id and ff.feedid = feed.id
  and ff.friend.id = f.id
```

We might attempt to support this type of query in GAE by adding a `feedCategories` multi-valued property list to `Friend`, which contained all the categories in which their feeds fell. Every time a feed was added to the `Friend`, this list would be updated with the new category as necessary. We could then perform a JDQL query to find all such Friends:

```
select from Friend where feedCategories == 'PR'
```

However, for each returned `Friend` we would then need to check each of their feeds in turn to determine which feed(s) were the PR ones—requiring further Datastore access.

To address this, we could build a `Friend` `feedCategories` multi-valued property list whose ordering was synchronized with the `urls` list ordering, with the *nth* position in the categories list indicating the category of the *nth* feed. For example, suppose that `url1` and `url3` are of category 'PR', and `url2` is of category 'Technical'. The two lists would then be sorted as follows:

```
urls =  [url1, url2, url3, … ]

feedCategories = [PR, TECHNICAL, PR, …]
```

(For efficiency, we would probably map the categories to integers). Then, for each `Friend` returned from the previous query, we could determine which feed URLs were the 'PR' ones by their position in the feed list, without requiring further Datastore queries. In the previous example, it would be the URLs at positions 0 and 2—`url1` and `url3`.

This technique requires more expense at write time, in exchange for more efficient queries at read time. The approach is not always applicable—for example, it requires a one-to-one mapping between the items in the synchronized property lists, but can be very effective when it does apply.

Using transactions

As the App Engine documentation states,

A transaction is a Datastore operation or a set of Datastore operations that either succeed completely, or fail completely. If the transaction succeeds, then all of its intended effects are applied to the Datastore. If the transaction fails, then none of the effects are applied.

The use of transactions can be the key to the stability of a multiprocess application (such as a web app) whose different processes share the same persistent Datastore. Without transactional control, the processes can overwrite each other's data updates midstream, essentially stomping all over each other's toes. Many database implementations support some form of transactions, and you may be familiar with RDBMS transactions. App Engine Datastore transactions have a different set of requirements and usage model than you may be used to.

First, it is important to understand that a "regular" Datastore write on a given entity is *atomic*—in the sense that if you are updating multiple fields in that entity, they will either all be updated, or the write will fail and none of the fields will be updated. Thus, a single update can essentially be considered a (small, implicit) transaction—one that you as the developer do not explicitly declare. If one single update is initiated while another update on that entity is in progress, this can generate a "concurrency failure" exception. In the more recent versions of App Engine, such failures on single writes are now retried transparently by App Engine, so that you rarely need to deal with them in application-level code.

However, often your application needs stronger control over the atomicity and isolation of its operations, as multiple processes may be trying to read and write to the same objects at the same time. Transactions provide this control.

For example, suppose we are keeping a count of some value in a "counter" field of an object, which various methods can increment. It is important to ensure that if one Servlet reads the "counter" field and then updates it based on its current value, no other request has updated the same field between the time that its value is read and when it is updated. Transactions let you ensure that this is the case: if a transaction succeeds, it is as if it were done in isolation, with no other concurrent processes 'dirtying' its data.

Another common scenario: you may be making multiple changes to the Datastore, and you may want to ensure that the changes either all go through atomically, or none do. For example, when adding a new `Friend` to a `UserAccount`, we want to make sure that if the `Friend` is created, any related `UserAcount` object changes are also performed.

While a Datastore transaction is ongoing, no other transactions or operations can see the work being done in that transaction; it becomes visible only if the transaction succeeds.

Additionally, queries inside a transaction see a consistent "snapshot" of the Datastore as it was when the transaction was initiated. This consistent snapshot is preserved even after the in-transaction writes are performed. Unlike some other transaction models, with App Engine, a within-transaction read after a write will still show the Datastore as it was at the beginning of the transaction.

Datastore transactions can operate only on entities that are in the same **entity group**. We discuss entity groups later in this chapter.

Transaction commits and rollbacks

To specify a transaction, we need the concepts of a transaction **commit** and **rollback**.

A transaction must make an explicit "commit" call when all of its actions have been completed. On successful transaction commit, all of the create, update, and delete operations performed during the transaction are effected atomically.

If a transaction is rolled back, none of its Datastore modifications will be performed. If you do not commit a transaction, it will be rolled back automatically when its Servlet exits. However, it is good practice to wrap a transaction in a `try/finally` block, and explicitly perform a rollback if the commit was not performed for some reason. This could occur, for example, if an exception was thrown.

If a transaction commit fails, as would be the case if the objects under its control had been modified by some other process since the transaction was started the transaction is automatically rolled back.

Example—a JDO transaction

With JDO, a transaction is initiated and terminated as follows:

```
import javax.jdo.PersistenceManager;
import javax.jdo.Transaction;
…
  PersistenceManager pm = PMF.get().getPersistenceManager();
  Transaction tx;
    …
  try {
     tx = pm.currentTransaction();
     tx.begin();
     // Do the transaction work
     tx.commit();
  }
  finally {
    if (tx.isActive()) {
       tx.rollback();
    }
  }
```

A transaction is obtained by calling the `currentTransaction()` method of the `PersistenceManager`. Then, initiate the transaction by calling its `begin()` method. To commit the transaction, call its `commit()` method. The `finally` clause in the example above checks to see if the transaction is still active, and does a `rollback` if that is the case.

While the preceding code is correct as far as it goes, it does not check to see if the commit was successful, and retry if it was not. We will add that next.

App Engine transactions use optimistic concurrency

In contrast to some other transactional models, the *initiation* of an App Engine transaction is never blocked. However, when the transaction attempts to commit, if there has been a modification in the meantime (by some other process) of any objects in the same entity group as the objects involved in the transaction, the transaction commit will fail. That is, the commit not only fails if the objects in the transaction have been modified by some other process, but also if any objects in its entity group have been modified. For example, if one request were to modify a `FeedInfo` object while its `FeedIndex` child was involved in a transaction as part of another request, that transaction would not successfully commit, as those two objects share an entity group.

App Engine uses an *optimistic concurrency* model. This means that there is no check when the transaction initiates, as to whether the transaction's resources are currently involved in some other transaction, and no blocking on transaction start. The commit simply fails if it turns out that these resources have been modified elsewhere after initiating the transaction. Optimistic concurrency tends to work well in scenarios where quick response is valuable (as is the case with web apps) but contention is rare, and thus, transaction failures are relatively rare.

Transaction retries

With optimistic concurrency, a commit can fail simply due to concurrent activity on the shared resource. In that case, if the transaction is retried, it is likely to succeed.

So, one thing missing from the previous example is that it does not take any action if the transaction commit did not succeed. Typically, if a commit fails, it is worth simply retrying the transaction. If there is some contention for the objects in the transaction, it will probably be resolved when it is retried.

```
PersistenceManager pm = PMF.get().getPersistenceManager();
  // ...
  try {
    for (int i =0; i < NUM_RETRIES; i++) {
      pm.currentTransaction().begin();
      // ...do the transaction work ...
      try {
        pm.currentTransaction().commit();
        break;
      }
```

```
          catch (JDOCanRetryException e1) {
            if (i == (NUM_RETRIES - 1)) {
              throw e1;
            }
          }
        }
      }
      finally {
        if (pm.currentTransaction().isActive()) {
          pm.currentTransaction().rollback();
        }
        pm.close();
      }
```

As shown in the example above, you can wrap a transaction in a retry loop, where `NUM_RETRIES` is set to the number of times you want to re-attempt the transaction. If a commit fails, a `JDOCanRetryException` will be thrown. If the commit succeeds, the `for` loop will be terminated.

If a transaction commit fails, this likely means that the Datastore has changed in the interim. So, next time through the retry loop, be sure to start over in gathering any information required to perform the transaction.

Transactions and entity groups

An entity's **entity group** is determined by its key. When an entity is created, its key can be defined as a child of another entity's key, which becomes its *parent*. The child is then in the same entity group as the parent. That child's key could in turn be used to define another entity's key, which becomes *its* child, and so on. An entity's key can be viewed as a path of ancestor relationships, traced back to a *root* entity with no parent. Every entity with the same root is in the same entity group. If an entity has no parent, it is its own root.

Because entity group membership is determined by an entity's key, and the key cannot be changed after the object is created, this means that **entity group membership cannot be changed** either.

As introduced earlier, a transaction can only operate on entities from the same entity group. If you try to access entities from different groups within the same transaction, an error will occur and the transaction will fail.

You may recall from *Chapter 5* that in App Engine, JDO owned relationships place the parent and child entities in the same entity group. That is why, when constructing an owned relationship, you cannot explicitly persist the children ahead of time, but must let the JDO implementation create them for you when the parent is made persistent. JDO will define the keys of the children in an owned relationship such that they are the child keys of the parent object key. This means that the parent and children in a JDO owned relationship can always be safely used in the same transaction. (The same holds with JPA owned relationships).

So in the *Connectr* app, for example, you could create a transaction that encompasses work on a `UserAccount` object and its list of `Friends`—they will all be in the same entity group. But, you could not include a `Friend` from a different `UserAccount` in that same transaction—it will not be in the same entity group.

This App Engine constraint on transactions—that they can only encompass members of the same entity group—is enforced in order to allow transactions to be handled in a scalable way across App Engine's distributed Datastores. Entity group members are always stored together, not distributed.

Creating entities in the same entity group

As discussed earlier, one way to place entities in the same entity group is to create a JDO owned relationship between them; JDO will manage the child key creation so that the parent and children are in the same entity group.

To explicitly create an entity with an entity group parent, you can use the App Engine `KeyFactory.Builder` class. This is the approach used in the `FeedIndex` constructor example shown previously. Recall that you cannot change an object's key after it is created, so you have to make this decision when you are creating the object.

Your "child" entity must use a primary key of type `Key` or String-encoded Key (as described in *Chapter 5*); these key types allow parent path information to be encoded in them. As you may recall, it is required to use one of these two types of keys for JDO owned relationship children, for the same reason.

If the data class of the object for which you want to create an entity group parent uses an app-assigned string ID, you can build its key as follows:

```
// you can construct a Builder as follows:
KeyFactory.Builder keyBuilder =
  new KeyFactory.Builder(Class1.class.getSimpleName(),
parentIDString);

// alternatively, pass the parent Key object:
Key pkey = KeyFactory.Builder keyBuilder =
  new KeyFactory.Builder(pkey);

// Then construct the child key
 keyBuilder.addChild(Class2.class.getSimpleName(), childIDString);
 Key ckey = keyBuilder.getKey();
```

Create a new `KeyFactory.Builder` using the key of the desired parent. You may specify the parent key as either a `Key` object or via its entity name (the simple name of its class) and its app-assigned (String) or system-assigned (numeric) ID, as appropriate. Then, call the `addChild` method of the `Builder` with its arguments—the entity name and the app-assigned ID string that you want to use. Then, call the `getKey()` method of `Builder`. The generated child key encodes parent path information. Assign the result to the child entity's key field. When the entity is persisted, its entity group parent will be that entity whose key was used as the parent.

This is the approach we showed previously in the constructor of `FeedIndex`, creating its key using its parent `FeedInfo` key.

See `http://code.google.com/appengine/docs/java/javadoc/com/google/appengine/api/datastore/KeyFactory.Builder.html` for more information on key construction.

If the data class of the object for which you want to create an entity group parent uses a system-assigned ID, then (because you don't know this ID ahead of time), you must go about creating the key in a different way. Create an additional field in your data class for the parent key, of the appropriate type for the parent key, as shown in the following code:

```
   @PrimaryKey
   @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
   private Key key;
   …

   @Persistent
     @Extension(vendorName="datanucleus", key="gae.parent-pk",
       value="true")
     private String parentKey;
```

Assign the parent key to this field prior to creating the object. When the object is persisted, the data object's primary key field will be populated using the parent key as the entity group parent. You can use this technique with any child key type.

Getting the entity parent key from the child key

Once a "child" key has been created, you can call the `getParent()` method of `Key` on that key to get the key of its entity group parent. You need only the key to make this determination; it is not necessary to access the actual entity. `getParent()` returns a `Key` object. We used this technique earlier in the *Splitting a model by creating an "index" and a "data" entity* section.

If the primary key field of the parent data class is an app-assigned string ID or system-assigned numeric ID, you can extract that value by calling the `getName()` or `getID()` method of the `Key`, respectively.

You can convert to/from the `Key` and String-encoded `Key` formats using the `stringToKey()` and `keyToString()` methods of the `KeyFactory`.

Entity group design considerations

Entity group design considerations can be very important for the efficient execution of your application.

Entity groups that get too large can be problematic, as an update to any entity in a group while a transaction on that group is ongoing will cause the transaction's commit to fail. If this happens a lot, it affects the throughput. But, you do want your entity groups to support transactions useful to your application. In particular, it is often useful to place objects with parent/child semantics into the same entity group. Of course, JDO does this for you, for its owned relationships. The same is true for JPA's version of owned relationships.

It is often possible to design relatively small entity groups, and then use **transactional tasks** to achieve required coordination between groups. Transactional tasks are initiated from within a transaction, and are enqueued only if the transaction commits. However, they operate outside the transaction, and thus it is not required that the objects accessed by the transactional task be in the transaction's entity group. We discuss and use transactional tasks in the *Transactional tasks* section of this chapter.

Also, it can be problematic if too many requests (processes) are trying to access the same entity at once. This situation is easy to generate if, for example, you have a counter that is updated each time a web page is accessed. Contention for the shared object will produce lots of transaction retries and cause significant slowdown. One solution is to **shard** the counter (or similar shared object), creating multiple counter instances that are updated at random and whose values are aggregated to provide the total when necessary. Of course, this approach can be useful for objects other than counters.

We will not discuss entity sharding further here, but the App Engine documentation provides more detail: `http://code.google.com/appengine/articles/sharding_counters.html`.

What you can do inside a transaction

As discussed earlier, a transaction can operate only over data objects from the same entity group. Each entity group has a root entity. If you try to include multiple objects with different root entities within the same transaction, it is an error.

Within a transaction, you can operate only on (include in the transaction) those entities obtained via retrieval by key, or via a query that includes an ancestor filter. That is, only objects obtained in those ways will be under transactional control. Queries that do not include an ancestor filter may be performed within the transactional block of code without throwing an error, but the retrieved objects will not be under transactional control.

Queries with an "ancestor filter" restrict the set of possible hits to objects only with the given ancestor, thus restricting the hits to the same entity group. JDO allows querying on "entity group parent key fields", specified as such via the `@Extension(vendorName="datanucleus", key="gae.parent-pk", value="true")` annotation, as a form of ancestor filter. See the JDO documentation for more information.

So, you may often want to do some preparatory work to find the keys of the object(s) that you wish to place under transactional control, then initiate the transaction and actually fetch the objects by their IDs within the transactional context. Again, if there are multiple such objects, they must all be in the same entity group.

As discussed earlier, after you have initiated a transaction, you will be working with a consistent "snapshot" of the Datastore at the time of transaction initiation. If you perform any writes during the transaction, the snapshot will not be updated to reflect them. Any subsequent reads you make within the transaction will still reflect the initial snapshot and will not show your modifications.

When to use a transaction

There are several scenarios where you should be sure to use a transaction. They should be employed if:

- You are changing a data object field relative to its current value. For example, this would be the case if you are modifying (say, adding to) a list. It is necessary to prevent other processes from updating that same list between the time you read it and the time you store it again. In our app, for example, this will be necessary when updating a `Friend` and modifying its list of `urls`.

  In contrast, if a field update is not based on the field's current value, then by definition, it suggests that you don't care if/when other updates to the field occur. In such a case, you probably don't need to employ a transaction, as the entity update itself is guaranteed to be atomic.
- You want operations on multiple entities to be treated atomically. That is, there are a set of modifications for which you want them all to happen, or none to happen. In our app, for example, we will see this when creating a `Friend` in conjunction with updating a `UserAccount`. Similarly, we want to ensure that a `FeedInfo` object and its corresponding `FeedIndex` are created together.
- You are using app-assigned IDs and you want to either create a new object with that ID, or update the existing object that has that ID. This requires a test and then a create/update, and these should be done atomically. In our app, for example, we will see this when creating feed objects, whose IDs are URLs.

  If you are using system-assigned IDs, then this situation will not arise.
- You want a consistent "snapshot" of state in order to do some work. This might arise, for example, if you need to generate a report with mutually consistent values.

Adding transactional control to the *Connectr* application

We are now equipped to add transactional control to the *Connectr* application.

First, we will wrap transactions around the activities of creating, deleting, and modifying `Friend` objects. We will also use transactions when creating feed objects: because `FeedInfo` objects use an app-assigned ID (the feed URL string), we must atomically test whether that ID already exists, and if not, create a new object and its associated `FeedIndex` object. We will also use transactions when modifying the list of `Friend` keys associated with a feed, as this is a case where we are updating a field (the `friendKeys` list) relative to its current value.

We will not place the feed content updates under transactional control. The content updates are effectively idempotent—we don't really care if one content update overwrites another—and we want these updates to be as quick as possible.

However, there is a complication with this approach. When we update a `Friend`'s data, their list of `urls` may change. We want to make any accompanying changes to the relevant feed objects at the same time—this may involve creating or deleting feed objects or editing the `friendKey` lists of the existing ones.

We'd like this series of operations to be under control of the same transaction, so that we don't update the `Friend`, but then fail to make the required feed object changes. But they can't be—a `Friend` object is not in the same entity group as the feed objects. The `Friend` and feed objects have a many-to-many relationship and we do not want to place them all in one very large entity group.

App Engine's **transactional tasks** will come to our rescue.

Transactional tasks

App Engine supports a feature called **transactional tasks**. We have already introduced some use of tasks and Task Queues in *Chapter 7* (and will return to the details of task configuration in *Chapter 12*). Tasks have specific semantics in the context of a transaction.

> If you add a task to a Task Queue within the scope of a transaction, that task will be enqueued *if and only if* the transaction is successful.

A transactional task will not be placed in the queue and executed unless the transaction successfully commits. If the transaction rolls back, the task will not be run. The specified task is run outside of the transaction and is not restricted to objects in the same entity group as the original transaction. At the time of writing, you can enqueue up to five transactional tasks per transaction.

You may recall from *Chapter 7* that once a task is enqueued and executed, if it returns with error status, it is re-enqueued to be retried until it succeeds. So, you can use transactional tasks to ensure that either all of the actions across multiple entity groups are eventually performed or none are performed. There will be a (usually small) period of time when the enqueued task has not yet been performed and thus not all of the actions are completed, but eventually, they will all (or none) be done. That is, transactional tasks can be used to provide **eventual consistency** across more than one entity group.

We will use transactional tasks to manage the updates, deletes, and adds of feed URLs when a `Friend` is updated. We place the changes to the `Friend` object under transactional control. As shown in the following code from `server.FriendsServiceImpl`, within that transaction we enqueue a transactional task that performs the related feed object modifications. This task will be executed if and only if the transaction on the `Friend` object commits.

```
@SuppressWarnings("serial")
public class FriendsServiceImpl extends RemoteServiceServlet
  implements FriendsService {

  private static final int NUM_RETRIES =5;
  …
  public FriendDTO updateFriend(FriendDTO friendDTO){

    PersistenceManager pm = PMF.getTxnPm();
    if (friendDTO.getId() == null) { // create new
      Friend newFriend = addFriend(friendDTO);
      return newFriend.toDTO();
    }

    Friend friend = null;
    try {
     for (int i=0;i<NUM_RETRIES;i++) {
       pm.currentTransaction().begin();
       friend = pm.getObjectById(Friend.class,friendDTO.getId());

       Set<String> origurls = new HashSet<String>(friend.getUrls());

       // delete feed information from feedids cache
       // we only need to do this if the URLs set has changed...
       if (!origurls.equals(friendDTO.getUrls())) {
         CacheSupport.cacheDelete(feedids_nmspce,friendDTO.getId());
       }

       friend.updateFromDTO(friendDTO);

       if (!(origurls.isEmpty() && friendDTO.getUrls().isEmpty())) {
         // build task payload:
         Map<String,Object> hm = new HashMap<String,Object>();
         hm.put("newurls", friendDTO.getUrls());
         hm.put("origurls", origurls);
         hm.put("replace", true);
         hm.put("fid",friendDTO.getId());
```

```
            byte[]data = Utils.serialize(hm);

            // add transactional task to update the
            // url information
            // the task will not be run if the
            // transaction does not commit.
            Queue queue = QueueFactory.getDefaultQueue();
            queue.add(url("/updatefeedurls").payload(data,
              "application/x-java-serialized-object"));
          }
          try {
            pm.currentTransaction().commit();
            logger.info("in updateFriend, did successful commit");
            break;
          }
          catch (JDOCanRetryException e1) {
            logger.warning(e1.getMessage());
            if (i==(NUM_RETRIES-1)) {
              throw e1;
            }
          }
        }// end for
      } catch (Exception e) {
        logger.warning(e.getMessage());
        friendDTO = null;
      } finally {
        if (pm.currentTransaction().isActive()) {
          pm.currentTransaction().rollback();
          logger.warning("did transaction rollback");
          friendDTO = null;
        }
        pm.close();
      }
      return friendDTO;
    }
```

The previous code shows the `updateFriend` method of `server.FriendsServiceImpl`. Prior to calling the transaction commit, a task (to update the feed URLs) is added to the default Task Queue. The task won't be actually enqueued unless the transaction commits. A similar model, including the use of a transactional task, is employed for the `deleteFriend` and `addFriend` methods of `FriendsServiceImpl`.

We configure the task with information about the "new" and "original" `Friend urls` lists, which it will use to update the feed objects accordingly. Because this task requires a more complex set of task parameters than we have used previously, we convert the task parameters to a `Map`, and pass the `Map` in serialized form as a `byte[]` task payload. See the *Task Parameters: Sending a Payload of byte[] Data as the Request* section of this chapter for more information about payload generation and use.

This task is implemented by a Servlet, `servlet,server.servlets.UpdateFeedUrlsServlet`, which is accessed via `/updatefeedurls`. This Servlet covers the three cases of URL list modification—adds, deletes, and updates.

The following code is from the `server.servlets.UpdateFeedUrlsServlet` class.

```
@SuppressWarnings("serial")
public class UpdateFeedUrlsServlet extends HttpServlet {
  …
  public void doPost(HttpServletRequest req, HttpServletResponse
     resp) throws IOException {

    Set<String> badurls = null;
    // deserialize the request
    Object o = Utils.deserialize(req);
    Map<String,Object> hm = (Map<String, Object>) o;
    Set<String> origurls = (Set<String>)hm.get("origurls");
    Set<String> newurls = (Set<String>)hm.get("newurls");
    Boolean replace = (Boolean)hm.get("replace");
    Boolean delete = (Boolean)hm.get("delete");
    String fid = (String)hm.get("fid");
   …

    if (delete != null && delete) {
      if (origurls != null) {
         FeedIndex.removeFeedsFriend(origurls, fid);
      }
       else {
         return;
       }
    }
     …
  }
}
```

The previous code shows the initial portion of the doPost method of UpdateFeedUrlsServlet. It shows the Servlet deserialization of the request, which holds the task payload. The resulting Map is used to define the task parameters. If the delete flag is set, the origurls Set is used to indicate the FeedIndex objects from which the Friend key (fid) should be removed.

Similarly, the following code shows the latter portion of the doPost method of server.servlets.UpdateFeedUrlsServlet. If the task is not a deletion request, then depending upon how the different task parameters are set, either the FeedIndex.addFeedURLs method or the FeedIndex.updateFeedURLs method is called.

```
@SuppressWarnings("serial")
public class UpdateFeedUrlsServlet extends HttpServlet {

  private static final int NUM_RETRIES = 5;
  public void doPost(HttpServletRequest req, HttpServletResponse
   resp) throws IOException {

      …
    if (origurls == null) {
      // then add only -- no old URLs to deal with
      badurls = FeedIndex.addFeedURLs(newurls, fid);
    }
    else { //update
      badurls = FeedIndex.updateFeedURLs(newurls, origurls, fid,
       replace);
    }
    if (!badurls.isEmpty()) {
     // then update the Friend to remove those bad urls from its set.
     // Perform this operation in a transaction
      PersistenceManager pm = PMF.getTxnPm();
      try {
        for (int i =0; i < NUM_RETRIES; i++) {
          pm.currentTransaction().begin();
          Friend.removeBadURLs(badurls, fid, pm);
          try {
            pm.currentTransaction().commit();
            break;
          }
          catch (JDOCanRetryException e1) {
           if (i == (NUM_RETRIES -1)) {
             throw e1;
           } } } }
      finally {
       if (pm.currentTransaction().isActive()) {
            pm.currentTransaction().rollback();
            logger.warning("did transaction rollback");
          }
          pm.close();
        } } }
    }
  }
```

In the case where URLs were added to the `urls` list of a `Friend`, some of the new URLs may have been malformed, or their endpoints unresponsive. In this case, they are returned as `badurls`. It is necessary to update the `Friend` object with this information—specifically, to remove any bad URLs from its `urls` list. This operation is performed in a transaction within the task Servlet. As with previous examples, the transaction may be retried several times if there are commit problems.

What if something goes wrong during the feed update task?

In `UpdateFeedUrlsServlet`, in addition to the `Friend` transaction, the operations on the feed objects are using transactions under the hood (in the `FeedIndex` methods).

Because all the transactions initiated by the task may be retried multiple times, it is quite unlikely that any of them will not go through eventually. However, it is not impossible that one might fail to eventually commit. So, we want to consider what happens if a failure were to take place.

The `FeedIndex` operations can be repeated multiple times without changing their effect, as we are just adding and removing specified friend keys from the sets of keys. So, if the task's `Friend` update transaction in the example above fails after its multiple retries, the Servlet will throw an exception and the entire task will be retried until it succeeds (recall that if a task returns an error, it is automatically retried). It is okay to redo the task's feed operations, so this scenario will cause no problems.

If a `FeedIndex` URL add/update transaction fails after multiple retries, this will result in that URL being marked as "bad". So, no information in the system ends up as inconsistent, though the client user will have to re-enter the "bad" URL.

If a `FeedIndex` URL delete transaction fails after multiple retries, it will throw an exception, which will result in the task being retried until it succeeds. Again, this causes no problems.

So, even if the transactions performed in the task themselves fail to go through on the first invocation of the task, the system will nevertheless reach an eventually consistent state.

Task parameters—sending a payload of byte[ ] data as the request

The previous code, from `FriendsServiceImpl`, showed a task using a `byte[]` "payload" as specification, rather than the individual string params that we had employed in *Chapter 7*. The following syntax is used to specify a payload, where `data` refers to a `byte[]`:

```
Queue queue = QueueFactory.getDefaultQueue();
queue.add(url("/updatefeedurls").payload(data,
 "application/x-java-serialized-object"));
```

When the task is invoked, the task Servlet's request parameter, `req`, will hold the payload data and may be deserialized from the request byte stream back into its original object, for example, as shown in the previous code, `UpdateFeedUrlsServlet`. This can be a useful technique when the task parameters are too complex to easily deal with as Strings. In our case, we want to pass `Sets` as parameters. So for the `UpdateFeedUrlsServlet` task, a `Map` containing the various task parameters is constructed and used for the payload, then deserialized in the task Servlet. Methods of `server.utils.Utils.java` support the serialization and deserialization. As an alternative approach, you could also pass as task params the identifier(s) of Datastore objects containing the parameter information. The task would then load the information from the Datastore when it is executed.

[Due to a GAE issue at the time of writing, base64 encoding and decoding is necessary for successful (de)serialization; this is done in `server.utils.Utils` (thanks to a post by Vince Bonfanti for this insight).]

Using Memcache

Memcache is one of the App Engine *services*. It is a volatile-memory key-value store. It operates over all your JVM instances; as long as an item remains in Memcache, it can be accessed by any of your application's processes.

Memcache contents remain indefinitely if you don't set them to expire, but can be removed ("evicted") by App Engine at any time. So, never count on a Memcache entry to exist in order for your application to work correctly. The service is meant only as a cache that allows you quicker access to information that you would otherwise obtain from the Datastore or have to generate. Memcache is often used both for storing copies of data objects and for storing relatively static display information, allowing web pages to be built more quickly.

Transactions over Memcache operations are not supported—any Memcache changes you make within a transaction are not undone if the transaction is rolled back.

The basic Memcache operations are `put`, `get`, and `delete`: `put` stores an object in the cache indexed by a key; `get` accesses an object based on its cache key; and `delete` removes the object stored at a given cache key. A Memcache `put` is *atomic*—the entire object will be stored properly or not at all. Memcache also has the ability to perform increments and decrements of cache values as atomic operations.

Objects must be serializable in order to be stored in Memcache. At the time of writing, Memcache has a 1MB size limit on a given cached value, and data transfer to/from Memcache counts towards an app quota (we further discuss quotas in *Chapter 11*).

Memcache has two features that can be particularly useful in organizing your cached data—the ability to define cache namespaces and the ability to define when a cache entry expires. We'll use these features in *Connectr*.

App Engine supports two different ways to access Memcache—via an implementation of the JCache API or via an App Engine Java API. JCache is a (not-yet-official) proposed interface standard, JSR 107.

The *Connectr* app will use the App Engine's Memcache API, which exposes a bit more of its functionality.

> This page and its related links have more information on uses for Memcache: `http://code.google.com/appengine/articles/scaling/memcache.html`.
>
> For more information on JCache, see `http://jcp.org/en/jsr/detail?id=107`and `http://code.google.com/appengine/docs/java/memcache/usingjcache.html`.

Using the App Engine Memcache Java API in *Connectr*

To access the Memcache service in *Connectr*, we will use the `com.google.appengine.api.memcache` package (`http://code.google.com/appengine/docs/java/javadoc/com/google/appengine/api/memcache/package-summary.html`). To facilitate this, we'll build a "wrapper" class, `server.utils.cache.CacheSupport`, which does some management of Memcache namespaces, expiration times, and exception handling. The code for the `server.utils.cache.CacheSupport` class is as follows:

```
import com.google.appengine.api.memcache.Expiration;
import com.google.appengine.api.memcache.MemcacheService;
import com.google.appengine.api.memcache.MemcacheServiceException;
import com.google.appengine.api.memcache.MemcacheServiceFactory;

public class CacheSupport {

  private static MemcacheService cacheInit(String nameSpace){
   MemcacheService memcache =
    MemcacheServiceFactory.getMemcacheService(nameSpace);
   return memcache;
  }

  public static Object cacheGet(String nameSpace, Object id){
    Object r = null;
    MemcacheService memcache = cacheInit(nameSpace);
    try {
      r = memcache.get(id);
    }
    catch (MemcacheServiceException e) {
     // nothing can be done.
    }
   return r;
  }

  public static void cacheDelete(String namespace, Object id){
    MemcacheService memcache = cacheInit(nameSpace);
    memcache.delete(id);
  }

  public static void cachePutExp(String nameSpace, Object id,
   Serializable o, int exp)  {
    MemcacheService memcache = cacheInit(nameSpace);
    try {
      if (exp>0) {
        memcache.put(id, o, Expiration.byDeltaSeconds(exp));
      }
     else {
       memcache.put(id, o);
     }
    }
    catch (MemcacheServiceException e) {
      // nothing can be done.
    }
  }

  public static void cachePut(String nameSpace, Object id,
    Serializable o){
    cachePutExp(nameSpace, id, o, 0);
  }
}
```

As seen in the `cacheInit` method, to use the cache, first obtain a handle to the Memcache service via the `MemcacheServiceFactory`, optionally setting the **namespace** to be used:

```
MemcacheService memcache =
  MemcacheServiceFactory.getMemcacheService(nameSpace);
```

Memcache **namespaces** allow you to partition the cache. If the namespace is not specified, or if it is reset to null, a default namespace is used.

Namespaces can be useful for organizing your cached objects. For example (to peek ahead to the next section), when storing copies of JDO data objects, we'll use the classname as the namespace. In this way, we can always use an object's app-assigned String ID or system-assigned Long ID as the key without concern for key clashes.

You can reset the namespace accessed by the Memcache handle at any time by calling:

```
memcache.setNamespace(nameSpace);
```

Once set for a Memcache handle, the given namespace is used for the Memcache API calls. Therefore, any subsequent gets, puts, or deletes via that handle will access that namespace.

As this book goes to press, a new Namespace API is now part of App Engine. The **Namespace API** supports *multitenancy*, allowing one app to serve multiple "tenants" or client organizations via the use of multiple namespaces to separate tenant data.

A number of App Engine service APIs, including the Datastore and Memcache, are now namespace-aware, and a namespace may be set using a new **Namespace Manager**. The `getMemcacheService()` method used in this chapter, if set with a namespace, will override the more general settings of the Namespace Manager. So, for the most part, you do not want to use these two techniques togetherâ€"that is, if you use the new Namespace API to implement multitenancy, do not additionally explicitly set Memcache namespaces as described in this chapter. Instead, leave it to the Namespace Manager to determine the broader namespace that you are using, and ensure that your cache keys are unique in a given "tenant" context. `http://code.google.com/appengine/docs/java/multitenancy/overview.html` provides more information about multitenancy.

To store an object in Memcache, call:

```
memcache.put(key, value);
```

where `memcache` is the handle to the Memcache service, and both the `key` and the `value` may be objects of any type. The `value` object must be serializable. The `put` method may take a third argument, which specifies when the cache entry expires. See the documentation for more information on the different ways in which expiration values can be specified.

To retrieve an object with a given key from Memcache, call:

```
    Object r = memcache.get(key);
```

where again `memcache` is the handle to the Memcache service. If the object is not found, `get` will return `null`.

To delete an object with a given key from Memcache, call:

```
    memcache.delete(key);
```

If these operations encounter a Memcache service error, they may throw a `MemcacheServiceException`. It is usually a good idea to just catch any Memcache-generated errors.

Thus, the `cacheGet`, `cacheDelete`, and `cachePut`/`cachePutExp` methods of `CacheSupport` create a namespace-specific handler based on their namespace argument, perform the specified operation in the context of that namespace, and catch any `MemcacheServiceExceptions` thrown. The `cachePutExp` method takes an expiration time, in seconds, and sets the cached object to expire accordingly.

`CacheSupport` requires the cache value argument to implement `Serializable` (if the wrapper class had not imposed that requirement, a `put` error would be thrown if the value were not `Serializable`).

Memcache error handlers

The default error handler for the Memcache service is the `LogAndContinueErrorHandler`, which just logs service errors instead of throwing them. The result is that service errors act like cache misses. So if you use the default error handler, `MemcacheServiceException` will in fact not be thrown. However, it is possible to set your own error handler, or to use the `StrictErrorHandler`, which will throw a `MemcacheServiceException` for any service error. See the `com.google.appengine.api.memcache` documentation (`http://code.google.com/appengine/docs/java/javadoc/com/google/appengine/api/memcache/package-summary.html`) for more information.

Memcache statistics

It is possible to access statistics on Memcache use. Using the `com.google.appengine.api.memcache` API, you can get information about things such as the number of cache hits and misses, the number and size of the items currently in the cache, the age of the least-recently accessed cache item, and the total size of data returned from the cache.

The statistics are gathered over the service's current uptime (and you can not explicitly reset them), but they can be useful for local analysis and relative comparisons.

Atomic increment/decrement of Memcache values

Using the `com.google.appengine.api.memcache` API, it is possible to perform atomic increments and decrements on cache values. That is, the read of the value, its modification, and the storage of the new value can be performed atomically, so that no other process may update the value between the time it is read and the time it is updated. Because Memcache operations cannot be a part of regular transactions, this can be a useful feature. For example, it can allow the implementation of short-term volatile-memory locks. Just remember that the items in the cache can be evicted by the system at any time, so you should not depend upon any Memcache content for the correct operation of your app.

The atomic increments and decrements are performed using the variants of the `increment()` and `incrementAll()` methods `of MemcacheService`. You specify the delta by which to increment and can perform a decrement by passing a negative delta. See the `com.google.appengine.api.memcache` documentation for more information.

Using Memcache with JDO data objects

One common use of the Memcache service is to cache copies of persistent data objects in volatile memory so that you don't always have to make a more time-consuming Datastore fetch to access them: you can check the cache for the object first, and only if you have a cache miss do you need to access the Datastore. Objects must be serializable in order to be stored in Memcache, so any such cached data classes must implement Serializable. When storing a JDO object in Memcache, you are essentially storing a detached copy, so be sure to prefetch any lazily loaded fields that you want to include in the cached object before storing it.

When using Memcache to cache data objects, be aware that there is no way to guarantee that related Memcache and Datastore operations always happen together—you can't perform the Memcache operations under transactional control, and a Memcache access might transiently fail (this is not common, but is possible), leaving you with stale cached data.

Thus, it is not impossible for a Memcache object to get out of sync with its Datastore counterpart. Typically, the speedup benefits of using Memcache far outweigh such disadvantages. However, you may want to give all of your cached objects an expiration date. This helps the cache "re-sync" after a period of time if there are any inconsistencies.

The pattern of cache usage for data objects is typically as follows, depending upon whether or not an object is being accessed in a transaction.

- **Within a transaction**

  When accessing an object from within a transaction, you should not use the cached version of that object, nor update the cache inside the transaction. This is because Memcache is not under transactional control. If you were to update the cache within a transactional block, and then the transaction failed to commit, the Memcache data would be inconsistent with the Datastore. So when you access objects inside a transaction, purge the cache of these objects.

  Post-transaction, you can cache a detached copy of such an object, once you have determined that the commit was successful.

- **Outside a transaction**

  If a Datastore access is not under transactional control, this means that it is not problematic to have multiple processes accessing that object at the same time. In that case, you can use Memcache as follows:

  When reading an object: first check to see if the object is in the cache; if not, then fetch it from the Datastore and add it to the cache.

  When creating or modifying an object: save it to the Datastore first, then update the cache if the Datastore operation was successful.

  When deleting an object: delete from the cache first, then delete from the Datastore.

In all cases, be sure to catch any errors thrown by the Memcache service so that they do not prevent you from doing your other work.

When using Memcache to store data objects, it can be useful to employ some form of caching framework, so that you do not have to add object cache management code for every individual method and access. In the next section, we will look at one way to do this—using capabilities provided by JDO.

JDO lifecycle listeners

JDO 2.0 defines an interface for the `PersistenceManager` (and `PersistenceManagerFactory`) that allows a listener to be registered for persistence events in the "lifecycle" of a data object—creating, deleting, loading, or storing the object. Once a listener is set up for a `PersistenceManager`, the JDO implementation will call methods on the listener when specified persistence events occur. This allows the application to define methods that monitor the persistence process (for specified persistent classes). We will use this feature to support object cache management in *Connectr*.

To set up a JDO lifecycle listener, create a class that implements any number of the following `javax.jdo.listener` interfaces—`DeleteLifecycleListener`, `CreateLifecycleListener`, `LoadLifecycleListener`, and `StoreLifecycleListener`. Each interface requires that the implementing class implements a "pre" and/or "post" method for the associated event type. These methods, often referred to as *callbacks*, are called before and after the persistence event. For example, `preDelete()` and `postDelete()` methods are defined for the `DeleteLifecycleListenerinterface`, which are called before and after a *delete* event.

Each method takes as its argument the `InstanceLifecycleEvent` event that triggered it. If triggered within a transaction, the listener actions are performed within the context of that same transaction.

See `http://www.datanucleus.org/products/accessplatform_1_1/jdo/lifecycle_listeners.html` for further details on Lifecycle listeners.

The following code shows one of the listener classes used in *Connectr*, `server.utils.cache.CacheMgmtTxnLifecycleListener`. It implements the `DeleteLifecycleListener` and `LoadLifecycleListener` interfaces, and thus is required to define methods called on *delete* and *load* events.

```
import javax.jdo.listener.DeleteLifecycleListener;
import javax.jdo.listener.InstanceLifecycleEvent;
import javax.jdo.listener.LoadLifecycleListener;


public class CacheMgmtTxnLifecycleListener implements
  DeleteLifecycleListener, LoadLifecycleListener {

  public void preDelete(InstanceLifecycleEvent event) {
    removeFromCache(event);
  }
```

```
    public void postDelete(InstanceLifecycleEvent event) {
      // must be defined even though not used
    }

    public void postLoad(InstanceLifecycleEvent event) {
      removeFromCache(event);
    }

    private void removeFromCache(InstanceLifecycleEvent event) {
      Object o = event.getSource();
      if(o instanceof Cacheable) {
        Cacheable f = (Cacheable) o;
        f.removeFromCache();
      }
    }
  }
```

Once a listener class has been defined, you can associate it with a given `PersistenceManager` instance as follows:

```
    pm.addInstanceLifecycleListener(
       new CacheMgmtTxnLifecycleListener(), classes);
```

where `classes` is an array of `java.lang.Class` objects for which changes should be listened for. That is, the `PersistenceManager` will listen for *delete* and *load* events on the given classes and call the defined listener methods when they occur.

Defining a cacheable interface

To use the JDO listeners to support Memcache management in *Connectr*, we'll first define a `server.utils.cache.Cacheable` interface:

```
  public interface Cacheable
  {
    public void addToCache();
    public void removeFromCache();
  }
```

Any data class that implements this interface must define two instance methodsâ€”an `addToCache()` method, which adds that object to Memcache, and a `removeFromCache()` method, which deletes that object from Memcache. The lifecycle listener methods (as in the previous example, `server.utils.cache.CacheMgmtTxnLifecycleListener`) can thus safely call `addToCache()` and `removeFromCache()` on any `Cacheable` object.

The implementations of these methods will in turn use the `server.utils.cache.CacheSupport` class (described previously) to actually access the cache, using the class name to set the Memcache *namespace*.

In *Connectr*, both the `Friend` and `FeedInfo` classes implement `Cacheable`. The following code shows the `server.domain.FeedInfo` implementation of `addToCache()` and `removeFromCache()` in support of that interface. For `FeedInfo`, the call to `getFeedInfo()` prior to adding the object to Memcache forces the feed content field to be fetched. Otherwise (because it is lazily loaded), that field would not be included when the object was serialized for Memcache.

```
@PersistenceCapable(identityType = IdentityType.APPLICATION,
   detachable="true")
public class FeedInfo implements Serializable, Cacheable {


  …
  @Persistent
  @PrimaryKey
  private String urlstring;
  …
  private int update_mins;


  public void removeFromCache() {
    CacheSupport.cacheDelete(this.getClass().getName(), urlstring);
  }

  public void addToCache() {
    getFeedInfo();
    CacheSupport.cachePut(this.getClass().getName(),urlstring,this);
  }
  …
 }
```

The *Connectr* application's lifecycle listeners

Any number of lifecycle listener classes can be defined, which implement handlers for some or all of the persistence events. For *Connectr*, we have defined two—one for the case where the `PersistenceManager` is being used to manage a transaction, and one for the case where there is non-transactional Datastore access. This simple distinction works for our application; more complex applications might use others.

As discussed earlier, with transactions, it is safest to conservatively remove cached information for any object that the transaction accesses. In that way, if the transaction does not commit, the cache will not be out of sync. The `server.utils.cache.CacheMgmtTxnLifecycleListener` class listed previously will be used for accesses inside transactions, and deletes the object from the cache after the object is loaded or before it is deleted from the Datastore. It does not add to the cache at any point.

If not operating within a transaction—for example, for a read—then the `server.utils.cache.CacheMgmtLifecycleListener` class, which follows, is used. It too deletes an object from the cache before it is deleted in the Datastore. However, this Listener caches an object after it is loaded or stored.

```
public class CacheMgmtLifecycleListener implements
  DeleteLifecycleListener, LoadLifecycleListener,
  StoreLifecycleListener {

  public void preDelete(InstanceLifecycleEvent event) {
    Object o = event.getSource();
    If (o instanceof Cacheable) {
      Cacheable f = (Cacheable) o;
      f.removeFromCache();
    }
  }

  public void postDelete(InstanceLifecycleEvent event) {
    // must be defined even though not used
  }

  public void postLoad(InstanceLifecycleEvent event) {
    addToCache(event);
  }

  public void preStore(InstanceLifecycleEvent event) {
  }

  public void postStore(InstanceLifecycleEventevent) {
   addToCache(event);
  }

  private void addToCache(InstanceLifecycleEvent event) {
    Object o = event.getSource();
    if (o instanceof Cacheable) {
      Cacheable f = (Cacheable) o;
      f.addToCache();
    }
  }
}
```

Using the lifecycle listeners consistently

In *Connectr*, we will use Memcache to cache objects of the `FeedInfo` and `Friend` (and child) data classes. In later chapters, as we further develop the app, we will make additional use of the cache as well. We can use our two JDO lifecycle listeners to impose consistency on our object caching: we can enforce the use of one of the two listeners with each `PersistenceManager` and set the listeners to apply only to the `FeedInfo` and `Friend` classes, so that the listeners are not being employed unnecessarily.

To facilitate this, we will modify the app's `server.PMF` singleton class (shown as follows) to define the list of data classes for which the app uses Listener-based caching. Then, we'll add two methods (`getTxnPm()` and `getNonTxnPm()`) that return a `PersistenceManager` with the appropriate lifecycle listener set. Now, instead of simply grabbing a generic `PersistenceManager`, we will obtain a configured one via `getTxnPm()` or `getNonTxnPm()`, depending upon whether or not the PersistenceManager will be used inside a transaction.

The following code shows the modified `server.PMF` class (in the `classes` array, replace 'packagepath' with your path).

```
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.PersistenceManager;


public final class PMF {

  private static final java.lang.Class[] classes =
    new java.lang.Class[]{
    packagepath.server.domain.FeedInfo.class,
    packagepath.server.domain.Friend.class};
  private static final PersistenceManagerFactory pmfInstance =
    JDOHelper.getPersistenceManagerFactory("transactions-optional");

    privatePMF(){
    }

  public static PersistenceManagerFactory get(){
     return pmfInstance;
  }

  public static PersistenceManager getNonTxnPm(){
    PersistenceManager pm = pmfInstance.getPersistenceManager();
    pm.addInstanceLifecycleListener(new CacheMgmtLifecycleListener(),
       classes);
    return pm;
  }

  public static PersistenceManager getTxnPm(){
    PersistenceManager pm = pmfInstance.getPersistenceManager();
    pm.addInstanceLifecycleListener(
       new CacheMgmtTxnLifecycleListener(), classes);
    return pm;
  }
}
```

Checking for a cache hit

Once the Listener-based mechanism is in place to add objects to the cache appropriately, we can check the cache on reads, given the ID of an object. If we get a cache *hit*, we use the cached copy of the object. If we have a cache *miss*, we simply load the object from the Datastore. Using the `CacheMgmtLifecycleListener`, a Datastore load invokes the `postLoad()` Listener method. This will cache the object, making it available for subsequent reads.

The following code illustrates this approach in the `getFriendViaCache()` method of `server.FriendsServiceImpl`.

```
@SuppressWarnings("serial")
public class FriendsServiceImpl extends RemoteServiceServlet
  implements FriendsService
{
   …
  private Friend getFriendViaCache(String id, PersistenceManager pm){
    Friend dsFriend = null, detached = null;

    // check cache first
    Object o = null;
     o = CacheSupport.cacheGet(Friend.class.getName(), id);
     if (o != null && o instanceof Friend) {
        detached = (Friend) o;
     }
     else {
      // the fetch will automatically add to cache via the lifecycle
      // listener
      dsFriend = pm.getObjectById(Friend.class, id);
      dsFriend.getDetails();
      detached = pm.detachCopy(dsFriend);
     }
     return detached;
  }
  …
 }
```

Memcache is not useful solely for data object caching. It is often used to cache display-related information as well. We will see this usage of Memcache in a later chapter.

Summary

In this chapter, we've explored ways in which you can make a GAE application more robust, responsive, and scalable in its interactions with the Datastore, and applied these techniques to our *Connectr* app.

We first took a look at how Datastore access configuration and data modeling can impact application efficiency and discussed some approaches to entity design towards scalability.

Then, we introduced transactions—what they are, the constraints on a transaction in App Engine, and when to use them.

Finally, we discussed the App Engine's Memcache service, how caching can speed up your app's server-side operations, and looked at an approach to integrating caching with access to "lifecycle events" on JDO data objects.