Persistent data

Interaction is fundamentally about changing information. In most cases that information is ultimately intended to be shared in some way. Small computer games may not share information but virtually every other interaction must. Persistence is the process of converting information into a form where it can persist outside the life of the application that is manipulating it. We want our information to persist across time and among people.

Issues

Before launching into various architectural solutions we first need an overview of the issues that must be addressed by our persistence architecture. These are:

- Pointers and files
- Create-mostly or read-mostly
- Bandwidth, latency and failure
- Format changes over time

Pointers and files

Most interactive data live in memory as data structures of various forms. In this book we have focused on trees for reasons that will serve us well in this chapter. Most data structures are based on pointers. Pointers are fundamentally memory addresses that do not have meaning outside of a particular running process. We will need representations that preserve the structure of the data while discarding the pointers.

Create-mostly to read-mostly

There is a continuum of interactive programs ranging from create-mostly through read-mostly. In the early days of the Macintosh and the IBM PC interactive applications were create-mostly. This includes word processors, spreadsheets, Powerpoint, computer aided design and many others. The purpose of these applications was to allow people to create information that they could share with other people for a variety of purposes.

With the advent of the Internet and the WWW has come a proliferation of read-mostly applications. This is where most of the interactive behavior is to search, retrieve, browse and otherwise experience information that already exists. Google is almost read-only. Except for entering search terms the entire interactive experience is browsing. There is no persistence problem at all. Amazon is read-mostly. Most of the interaction is searching the catalog of products with a limited amount of interaction to complete a purchase or manage your account.

Creation to reading is a range of applications rather than two categories. Many applications are mixtures of both. A word processing program (create) might be embedded in a document repository (read). There are many variations. This range of characteristics is helpful when we think about data persistence.

In a create-mostly application, the size of the information is limited by the human capacity to create that information. If someone could steadily type 100 words per minute for 2,000 hours per year (full time 40 hrs/week) they would generate 60 MB of data. Even if this were all one file it would be modest by today's computer memories. Of course nobody can sustain that kind of creative output. In reality individuals generate less than 1 MB per year (video excluded). Even if there are 10 – 100 people working on the same piece of work all year that is still less than 100 MB. The persistent data for create-mostly tasks tends to be relatively small and self-contained.

In a read-mostly application the data involved is huge (all of Google search, the Netflix video catalog, web browsing). There is no way that an interactive user can view more than a fraction at a time and certainly only modify a tiny fraction. This means that our interactive applications are mostly retrieving data and assuming that the data will not change during the lifetime of the interaction.

For create-mostly applications the persistence model is generally a set of self-contained documents or files. Each persists on its own and can be moved and shared as a single entity. Interaction generally occurs in RAM with persistence only coming into play when the file is saved or loaded. This model is breaking down somewhat with more and more applications continuously updating the persistent store rather than waiting for a “save” command. For read-mostly the persistence model is a database, generally accessed remotely over the internet with the interaction involving only a small portion at a time. Much of the interaction is for formulating search queries. Updating the store is based on a series of relatively small transactions. As we move forward more and more applications will occupy the middle ground between the two poles. Mobile devices with their limited interaction and memory capacities are driving towards this mixed approach.

Bandwidth, latency and failure

Applications which cannot interact over the internet are declining rapidly. Interacting “in the cloud” will be a key part of user interface architecture. With that power comes the issues of bandwidth, latency and failure. When one makes a request of a local hard disk there is high assurance that the information will be retrieved within the span of user patience and with an extremely high reliability. For a disk read to not work would be considered a system failure. These characteristics are not shared by the internet. When requesting information over a network, issues of latency and bandwidth will make exceeding user patience quite common. There are also periodic failures when the information is not retrieved. The latency and bandwidth issues are resolved by making information retrieval asynchronous with the user interface. The user interface cannot come to a halt while waiting for network retrieval. We need to modify our interactive architecture to account for this.

Failure to retrieve is a bigger challenge. This can happen for several reasons and can cause problems throughout. Our interactive application itself may fail. If our application fails it will just stop communicating with the server. The server needs to continue in a robust manner despite our failures. The network connection between our application and the server may fail. This leaves both sides with an empty silence that must be dealt with. We do not always get an error. Sometimes it is just silence. Sometimes the server will fail leaving our application without a data resource and sometimes a peer version of the application may fail. We need to cleanly account for all of these issues in our persistence architecture.

Lastly there is the need to minimize network round trips in our relations to the server. If on a given input event, we make a number of changes to our persistent model, we do not want a change message to be sent for every little change. Each such message breaks down into several synchronous messages across the internet. This is particularly true if we are using connectionless protocols like HTTP and error correcting protocols such as TCP. Numerous round trips with blocking behavior at each round trip will bring an interactive application to its knees. We need to batch up our changes so they can be transmitted in larger messages. The length of the message is less important than the number of messages in getting good interactive behavior.

Schema change

A schema is a definition of the structure of the persistent data. The word comes from the database community. The problem is that interactive applications change over time as users request new features. These changes must be reflected in the structure of the persistent data. The challenge comes when our application changes to accept data in a new form and our stored data has not changed. Such changes are essential to keep our customers buying new versions of our software. In our persistent data architecture, we must account for these changes.

Representation

Before we can create persistent data, we need a representation. We will consider three strategies here: serialization, relational tuples and key/value stores.

Serialization

In create-mostly applications, the entire work product is considered to be small enough to fit in RAM. This leads to the load/save model of persistence based on files. When a new work product is created an empty data structure is created in memory as the model of the user interface. As the user interacts this model is changed and generally grows as new information is added.

At appropriate times this model is saved to a file. The internal data structure with all of its pointers must be “serialized” into a stream of bytes that can be saved in the file system. The term serialization comes from the conversion of a graph-like data structure with pointers going in all directions into a sequential set of data with no pointers at all.

Historically a program would have a saveFile() function and a loadFile() function that would perform the serialization in whatever form seemed appropriate. Some languages such as Java provided automatic serialization features. Languages that support reflection have the ability to identify all of the fields in an object and save those fields. If the object has pointers, then it can follow the pointers and save those objects also. If one keeps track of which pointers are already saved, a pointer can be represented as a reference back to the first saving of that object.

One of the challenges of reflection-based serialization in strongly typed languages such as Java is schema change. If the type of an object is changed, Java serialization will no longer read older files. This becomes quite brittle, particularly during development when many things are changing. More dynamic data structures such as JavaScript can pretty much store anything. Data that is no longer used in a new incarnation can be safely ignored.

Serialization can also be binary or text. Binary is obviously small in space and faster in time. However, not all computers share the same binary representations. Language-based serialization overcomes this by adopting standard representations regardless of the hardware platform and then creates hardware-specific conversions.

On mobile devices the save/load model is being replaced by the continuous update approach. Whenever a change is made, it is automatically written out to the persistent representation. This makes the standard serialization approach less appealing.

Relational Tuples

Relational databases have long been a staple for large data stores. Their properties for indexing, flexible storage and access have been known for a long time. In this approach data is broken into tables of tuples. All tuples in the same table have the same set of fields. Tuples can reference other tuples using shared information. The standard operation for merging tuples from multiple tables is the Join operation.

The advantage of the relational model is that it has the least commitment to the actual use of the data. Data for a particular need is pulled together as necessary using the Join, Select, Project operations. When the desired data is viewed as a list of things, the resulting table view fits quite nicely. However, if the desired data is interrelated or tree structured in some way, the relational representation starts to break down. In the relational model the data is represented in a form that is suitable for long-lived databases but not necessarily for interactive needs.

Key/Value Store Model

A common representation in network-connected interactive application is the key/value store. It is then quite common for the value to be a tree. This structure was first introduced in chapter 1. This form can easily represent the relational tuple model where each key corresponds to a unique tuple and its value has the value of all that tuple's fields. In this representation the value can grow to the size of small serialized files. The representation can easily move back and forth.

There are two big advantages to the key value store: 1) it is easy to implement, manage and use, 2) it hides the data base structure while providing data in a form that is easy for the UI code to consume. The structure of the values is targeted towards simplifying the user interface code rather than just exposing the database structure. This separation also allows us to make major changes to the database without harming the user interface code. Throughout the rest of this chapter we will assume the use of the key/value store.

We still need a representation for the values in our store. Early on, XML was seen as a representation for such data values. That is where the AJAX (Asynchronous JavaScript and XML) architecture came to be. The problem is that XML stands for eXtensible Markup Language. Its heritage was in markup language for documents. The standards process started adding layers of constraint and type declaration onto the markup to constrain, declare and standardize what was there. In network-based inter action this grew into SOAP (Simple Object Access Protocol) which with its layers of specification, and discovery mechanisms is anything but simple. Systems have drifted towards much simpler representations such as JSON (JavaScript Object Notation). In JSON-like representations there is a clear, simple relationship between the textual representation and the data structure that it represents.

With smaller objects rather than entire files, the size of the representation is no longer as important. Thus, most data representations are text as in JSON or XML. The size of a network packet is not nearly as important to overall speed as the number of round trips. If necessary, text can be compressed and reach binary sizes or better. A huge advantage of a textual representation is in debugging. It is much easier to see what was sent when hunting down problems. If size in the database is a problem, then the database can choose its own representation. Our interactive application only cares about the representation used for communication.

Asynchrony

When so much of our persistent data is on the other side of a comparatively slow and error prone network, the issues of asynchrony manifest themselves in the user interface. There are two issues that are important to the user interface: 1) the user interface must not block while waiting for the network and 2) we always need to present the user with our best understanding of the state of the application.

We should never deprive the user of control of the program while waiting for the network. Too many interactive applications ignore these problems and leave the user stuck and powerless. This leads to very cranky users. We also must make clear what is going on. If there is information that has not yet arrived, that status must be visible. Taking an action and doing nothing visually leaves the user assuming that they did something wrong that they must somehow correct. If information is still loading or otherwise incomplete, that should be obvious to the user.

There are four basic mechanisms that we use in interactive applications to deal with asynchrony. They are: caching, promises, change notification and leases. Caching makes the UI more responsive by storing local copies of the data so that we do not need a network round-trip every time we want to update the presentation. Later in this chapter we will discuss how to do this so the results are reliable. When we make a request for an asynchronous data operation, there are three things that can happen: 1) the request eventually succeeds, 2) there is an error and the request fails, 3) after a reasonably period of time the request stops because there has been no response. Promises are a mechanism for dealing with these three outcomes. A promises are fulfilled or not, the local data must be updated and then change notification brings the presentation in line with whatever has happened to the model. To protect itself against unresponsive applications the server may only “lease” data for a fixed period of item. When the “lease” expires the data is no longer good and fresh data must be obtained from the server.

In discussing how our application should work with a persistent data model that functions asynchronously, there are three basic operations that are performed: 1) creating new objects, 2) deleting or changing existing objects, and 3) retrieving objects from the store. Each of these relates to the application and the data store in different ways.

Object Cache

If our application is going to have an object store that is asynchronously accessed, we will need a cache for the objects that we have received. Such a cache will have the following five operations:

- **Get** - retrieve an object given the key for that object.
- **Create** – create a new object and obtain a key for it.
- **Change/Delete** – modify an existing object.
- **Find** – search for one or more objects.
- **Save** – This will forward all changes known to the cache on to the data store.

We can define the function of our cache in terms of these four operations. These will also be sufficient to manage most of our asynchrony needs. It is very important that all accesses to our data store go through this cache. If we start to work outside of the cache to access the store directly, there will be problems. The cache mechanism also allows us to batch up changes and send them as a group to reduce network traffic.

Basic cache structure

Our cache is based on three hash tables and a counter, as shown in figure 17-1.

```
class DataStore
{   int objCounter=0;
    Hashtable recent=new Hashtable();
    Hashtable prev = new Hashtable();
    Hashtable dirty = new Hashtable();
}
```

17-1 – Data Store methods

An important part of our cache is that it only hold a limited number of items so that we do not overflow resources on our local device. For this we have a MAX_OBJECTS constant that establishes that limit. The idea of this cache is that recent will hold pointers to all objects that have been recently accessed. The prev table will hold objects that have been accessed, but not recently but for which we still have cached copies. The dirty table contains references to all objects that have changed since the last time the cache sent changes to the remote data store.

Get

The get() method takes a key and returns an object associated with that key. In strongly typed languages such as Java or C# there may be multiple get() methods, one for each type of object in the store. Such languages may also have get() return an Object which is the superclass of everything and then the application can sort out what object it has. Dynamically typed languages, such as JavaScript or Python, would have one get() method on the store. The code for get() is shown in figure 17-2.

```
Object get(String key)
{   Object rslt = recent.get(key);
    if (rslt == null)
    {   rslt = prev.get(key);
        if (rslt == null )
        {   rslt = a default object that indicates that
                the desired object is still loading from
                the store.
            Launch an asynchronous request for the object
        }
        recent.put(key,rslt);
        objCounter++;
        if (objCounter>MAX_OBJECTS)
        {   prev=recent;
            recent=new Hashtable();
            objCounter=0;
        }
    }
    return rslt;
}
```

17-2 – Getting objects from the cache

This method is given the key for the object to be retrieved. It first looks in the recent table and returns the object if it is found. If there has been no recent retrieval of the object, it looks in the prev table to see if the object is there. If the object is in prev then the object is entered into recent and the objCounter is incremented. This is a cheap mechanism for identifying recently used objects.

If the desired key is not in either of the two hash tables, then an asynchronous request is made to retrieve the object from the remote data store. The problem is that we do not want our user interface to block waiting for this request. The solution is to generate a default object that indicates that the real object has not yet arrived. This default object can be returned to the user interface and interaction can proceed. In the case of our employee, we may set all fields to null and put “waiting to load” in place of the name. This clearly indicates to the user that this data has not arrived. Alternatively we may put a flag on the object to indicate its temporary nature and then the UI can display that appropriately.

The UI code should place a listener (chapter 7) on the result of the get() so that it is notified of any changes. If the result was a default object, eventually the asynchronous request will complete, the object in the cache is updated and the listeners are notified. When the notification occurs, the UI can update its presentation to reflect the actual data now in the object. Our standard presentation listener mechanism will handle the asynchronous arrival of data from the store.

This process of promoting objects into the recent table cannot go on forever. We need to limit the size of the table. As seen in figure 17-2, when the number of objects exceeds MAX_OBJECTS, the recent table is moved to prev. The prev table is discarded and its objects are released to garbage collection. The recent table is then initialized to a fresh empty table. Initially the algorithm will retrieve objects from prev, but they will be steadily promoted into recent. Objects that are no longer used will stay in prev and eventually be discarded. This is not a strict least-recently-used cache discard algorithm but it is close enough and very simple to implement.

Create

When creating a new object there are really only three things that we need: the type of object to be created, a unique key for the object and a default initial value. The data store needs to know that we have created an object but there is nothing in the data store for that object. Most object data stores have multiple types of objects in the store. Our create method needs to know what type of object is to be created. In dynamically typed languages we can pass a string or integer parameter to indicate the type. In strongly typed languages we can create a separate create() method for each type of object to be created.

The key that identifies the object is the most important. We want a key that is unique across all uses of the data store. We also would not like the UI to wait on the data store before continuing. The answer to this is to prefetch a set of keys that are known to be unique. Suppose, for example that when a connection is first made with a data store, it returns 10 keys that it guarantees to be unique. When the user creates a new object the application takes one of those keys and assigns it to the new object. The object now has a unique key and there was no need to wait for the data store. The data store can be asynchronously notified that a new object was created under that key, but the UI can continue on without waiting. If the pool of potential keys falls below a threshold (5 for example) the application can request new keys from the data store. This guarantees that there are always plenty on hand so the application is never waiting for new keys.

The data store on the server must remember which keys it has handed out so that they are not inadvertently used by other instances of the application. The problem here is that applications may terminate while holding keys that they have not used. If the server leases the keys it has issued then eventually the lease will expire and the server can release those keys even though the application has not released them or used them.

If the key space is large, it is easy to create unique keys. Suppose we use a 10 character key that is randomly generated. If we include letters and numbers there are 36^{10} possible keys. If our random key generator is any good the likelihood of generating duplicate keys is extremely low. To be certain, the data store can generate a key and then perform a single key search (usually very fast). If the search fails, the key is unique. If the search succeeds, try again. It is virtually impossible for this to fail more than 2 times. Thus it is very easy for a server to quickly generate a batch of unique keys. The server must then remember the keys it generated until they are either used or their lease expires.

Delete/Change

Data objects change when some value is modified or deleted. For the purposes of this discussion changes and deletions are treated the same. The employee in figure 17-3 is an example of what needs to happen.

```
Employee{ key:”E1887”, lastName:”Jones”,
    firstName:”Andrew”,
    contact:{
        phones:[ “803-123-4567”, “991-543-1234” ],
        addresses: [
            {   street:”1014 Flingerhammer Dr.”,
                city:”Sholo”, state:”AK”
            },
            {   street:”742 Shootoss Av.”,
                city:”Limpopo”, state:”WA”
            }
        ]
    }
}
```

17-3 – Employee data

In figure 17-3 the key is shown as part of the employee record for convenience. It can just as easily be a separate part of the data store. Normally we would also want much longer keys because it is easier to create unique ones. Suppose our user has interactively deleted the second address (in Limpopo, WA). If we have created our models using the principles in chapter 2, the method that performed the deletion on the array of addresses can notify the addresses object of the change, which in turn can notify the contact object, which in turn can notify the Employee object, which is the root.

Using the principles in chapter 7 the data store cache can have posted itself as a listener on the employee object. Thus the cache is notified of the change and can respond accordingly. The presentation’s listeners are also notified and the display is updated. Alternatively root-level objects can always notify the cache of their changes so that the overhead of the listeners is not required. It is extremely rare that a root level model object will not want to notify the cache.

The notification of the cache of the changes can be handled in two ways. If the objects are relatively small, as in figure 17-3, we simply add the object to the dirty table so that the cache will get it updated when appropriate. The object has changed already so the presentation will reflect that change. If objects are relatively large, we can compute a change record, as shown in figure 17-4.

```
Delete{ path:["contact","addresses",1] }
```

17-4 – Change record for modifying 17-1

This change record can be sent to the cache and such changes can be collected into a batch to be set to the server all at one time. The cached object is still changed so the presentation is up-to-date. The asynchronous sending of changes can happen without impacting the user interface.

In some cases the change will fail. For example, the user may not have sufficient rights to delete an employee's address. In such case, the data store may reject the change. If that happens, the data store should resend its version of the object in question along with the rejection. The cached object is changed and the listeners are notified of the new object value. The presentation's listeners will restore the object to the state approved by the data store. The accompanying error can also be shared with the user.

So in response to a change in a persistent object the steps are:

- Change the cached object and notify that object's listeners, including the presentation.
- Mark the cached object or its change records to be sent to the server. (add object to the dirty table)
- In case of rejection, set the cache to the value received from the server and notify all presentation listeners of the change.

Search

It is very common to need to search for various kinds of objects in your data store. There are a wide variety of query languages in which such searches can be expressed. Google uses the simple list of words and returns the best matches. SQL is a sophisticated language for querying relational databases. There are other languages for querying JSON or HTML/XML markup data bases. For this discussion the query language does not matter. You will use the one that is best for your application and object store.

The issue that is special about search it that the results are almost always asynchronous, frequently do not arrive all at once and there are frequently too many to download into the application at one time. A further difficulty is the ordering of the results. The question is whether the object store should order the results or the application.

The first step is to treat each query as a special kind of object. They have a very short lifetime, but treating them as an object will simplify the relationship to the cache and to your presentation. In order to send a query you must create a new query object for which you need a key. This can be handled as described in the Create section above. The object is loaded with the query and put into the cache on the dirty list so that it will get sent with the next batch of requests. Listeners should also be attached and notified. This will allow the presentation to show the pending nature of the results. As results are received they will come with the key attached so that the results can be loaded into the correct query object and the listeners notified. As new requests related to this query are sent to the store, the key will identify the query to which the request belongs. Data stores frequently construct query objects that they retain as further results are requested and this key makes the connection possible for the data store. It is frequently helpful to provide a discard() method on query objects so that the server can be notified and free up its own resources.

Results of the query will be received asynchronously by your cache handler. The query object in the cache will be updated and listeners notified. Even if results come in piecemeal, the repeated listener notification will keep the presentation up-to-date with what has been received so far.

The results should be handled as an array or list of other objects. Because the array may be large and its results arriving asynchronously this array needs a special implementation. There generally three cases for such handling that depend on the number of expected results. If your array implementation can handle all three, the presentation need not know and can just present and interact with what is available already. Roughly we can thing of result sizes in tens, thousands and gazillions. In most cases this distinction can be drawn by the data store rather than the interactive application. The result array is added to the query object when it is created and then filled as results are returned. The result array is an array object keys. The application can then pull what it wants from the objects. And the cache and data store can handle object access through the normal mechanism.

Tens of results

If the result only has a few things, such as the classes that a student is taking, the children in a family or the pending orders in your Amazon account. In the returned result contains the entire array of keys and the objects referenced by those keys are also returned. Because of the small size of the result the array and the objects are easily batched into a single message. This preloads the cache with the objects so that the presentation can immediately show everything. It is also appropriate for the interactive application to do the ordering rather than the data server because all of the object information is present for the application to work with.

Hundreds to thousands of results

In this case there is too much data to appear on the screen at once and sending all of the objects along with the results will be way too large. The simplest approach is to send a results array with only the object keys. The presentation can then request the objects that it intends to show and then respond to notifications as those object contents arrive. A thousand objects with 20 byte keys (very large) would only take about 20K in a message. This is easily stored in RAM even on phones. The fact that it contains keys rather than objects lets the cache handle the rest of the details. If you are only looking at 20 objects at a time, the others will either not yet be retrieved or will be candidates for removal from the cache.

If the user scrolls back and forth a lot, objects that are viewed will stay in the cache and the user interface will be very responsive. When the presentation requests an object that is not in the cache it should receive an object that shows the pending status and to which listeners can be attached for notification when the results arrive.

Ordering of the results should be handled by the server rather than the application. Without all of the objects, the application is not capable of doing the sort. Downloading all of the objects just to do an application-side sort is not a good use of resources.

In many cases with thousands of results the data store will order them by likely interest. We see this in Facebook posts, Amazon product searches, Google results and many others. In such uses, the user frequently does not look beyond the first 10-20 results. In such situations the server can send the full array of keys along with the first 10-20 objects. This preloads the cache with the objects for the first view. The user instantly sees that first screen full of objects without 20 independent object requests or additional waiting. If the user scrolls or in some other way moves beyond that first group the object requests will bring in the new objects.

In the case where viewers are looking at objects from the beginning of the list forward, we can perform an optimization inside of our array. Whenever some part of the application accesses the object at location N in the array, we check to see if the objects for locations N through N-P are in the cache, if not the array requests them. P can be thought of as a page or prefetch constant. As the user works down through the list of results the array is prefetching objects that are not yet needed but might be. If P is too large then unnecessary numbers of objects are retrieved. If P is too small then objects may not have arrived when the presentation needs them and the user must wait. It is easy to adjust P based on actual usage.

Note that requests for objects go into the pending or dirty list maintained by the cache and are not actually issued until a save operation (discussed later). This allows multiple requests for the same object and requests for many objects to all be batched together in a single message.

Gazillions of Results

In many cases, such as Google searches or some product catalogs the number of results is huge and even the results array is too large to be shipped all at once. In some situations the length of the results array may not be known either because it is too costly to count or because the application wants to appear infinite. Facebook, for example, does not want you to see the end of a posting list. As long as you want to keep looking, they want to keep providing you information.

In this case the results array may have a length or it may not. Many Google searches will tell you how many results there are. (the length of the results array) For many searches it only gives you an estimate of its length. As discussed earlier, Facebook and Twitter want their feeds to appear infinite.

For such very large arrays the first results array has a length that matches the first batch of keys and then we allow the length to be updated as new objects are retrieved. Such very, very large lists are always ordered by user interest and their full extent is rarely retrieved interactively. For such large results we can use the prefetching technique described for thousands of results. We augment this slightly. If the results array is asked for the key at location N and the length of array is L and L>N-PL (where PL is our prefetch constant for length) then in addition to prefetching any objects, we also request an update on the results array length. PL is generally much larger than P. An increase in the results array will also prefetch the object keys that should fill those new slots in the array.

If application memory is tight or we expect our results lists to be long we can also implement our array so that it maintains a fixed number of object keys and just moves the starting index of the array. Thus when we ask for object keys to be added to the end, we discard object keys from the start of the array. This means that as we move close to that start index we will need to request the keys that we discarded earlier. The advantage of this is that memory demands in the application are fixed and user activity cannot cause memory overflow.

Save

In the preceding operations on the cache and objects in the cache all changes and requests were held in a “dirty” list or an update list. This is to prevent interactive behavior from generating clouds of redundant network requests. For example, any object already on the “dirty” list will not be added again. Thus many changes to an object will still only result in one entry for that object. A dirty list is a simple implementation. A log of change records is another alternative.

Eventually we need to send those requests to the data store so that they can be made available to the user. This is the save(), update() or flush()method on the cache. This tells the cache to send all pending requests to the data store and empty the dirty list or change log. The request results will arrive asynchronously and the listener mechanism will keep the presentation informed.

For interactive purposes, the save() method should generally be called at the end of every input event. This will make certain that the cache and data store start working on bringing the data store in line with the user’s requests. Exceptions to the “every input event” policy would be mouse movements. It is probably not a good use of resources to force data store updates on every mouse movement. It is generally a better policy to let such intermediate changes accumulate in the cache. In such cases one or a few data objects are being modified repeatedly. If save() is called on mouseUp or some other terminating event, all of the changes accumulated by the mouse movements will get sent in a timely manner without too much resource expenditure.

The invocation of save() can be a programming practice that is used throughout the application. The problem with this approach is that forgetting to call save() will leave the data store in an inconsistent state. In many cases this is resolved when a later event does call save() because the cache has not forgotten those changes. If some form of top-down event handling is being used, the top level event handler can take responsibility for calling save() after any of its descendants have processed the event. If top-down or object inheritance event handling is not available then the programming practice approach is the only choice.

Promises

Leases

Server/client partition of roles

Security

Schema change

Summary

Exercises