Development-Database vs Filesystem

Overview

There is an ongoing debate, generally related to media files, about whether to store media in the database or whether to store media on the fileystem and just store the reference in the database.

This article attempts to track some of the notable writings about the debate.

Joe Moreno

Keep in mind that the purpose of a database is to store data to be search and retrieved.

It would be a rare case when you'd actually send a query to a database that consisted of an image blob (i.e. search for an image that matches certain binary data). More than likely, you'd perform a search for an image based on its meta-data like date, time, image name, or file system path. A good solution is to store the path to the medium and then simply build the reference URL for the client's browser to reference or have the application retrieve the medium from the file system and serve it up through the WebObjects adaptor. In the former case you can keep the media under the Web server (say image thumbnails) and, in the latter case, you can keep full size images anywhere else on the server's file system and server them up based on a user's profile (i.e. did they successfully check out?, etc).

Michael Engelhart

Storing images in the database is generally a bad idea in my opinion. There's much more overhead in retrieving image data from a database then there is in just letting Apache serve up the image. Apache has been highly optimized just for this purpose. Databases generally have not.

My suggestion is to just store a URL for the image in the database and write that URL to the dynamic page. Or if you know the path is always going to be the same you could simply store the filename.

Robert Walker

It seems that different people have different opinions on this topic. I've followed several threads on this and I still haven't come to a conclusion on the best design pattern.

I wouldn't want to put the images in a share point on the network with a link to the image directory. That would create a single point of failure. If you lose the connection to the box storing the images, for any reason, then you have no access to the images. With images stored in the database you can use database clustering to provide some redundancy and fail-over support.

In either case you will be transporting the binary data for the images across the network either directly via a share, or through the database connection. I personally take the easy way out and store the images in the database.

An Example:
Say I have a Product entity and want to upload and store product photos: I would create two entities Product and ProductPhoto. I would then relate them with either a toOne or toMany relationship depending on whether I need one or many ProductPhoto objects for each Product object.

With this design pattern fetching Product data doesn't directly load the images. Instead EOF will create faults representing the images.
The image data isn't fetched until the fault is fired by accessing the ProductPhoto fault object. So If you fetch 500 Products and batch them into groups of 10 with the [WODisplayGroup|WO:Programming__WebObjects-Web Applications-Development-WODisplayGroup] then your first page would fetch only the first 10 images not the 500 (and only if there is a WOElement? or method that accesses the image data).

This pattern also greatly simplify uploading and storing the images because you can bind the NSData used to upload the image to your ProductPhoto's imageData BLOB.

It's probable that many will disagree with me on this issue, but I have had good success, for my purposes, with this design pattern.

You can find an implementation of this design pattern for both toOne and toMany photos in the JavaRealEstate framework example in /Developer/Examples/JavaWebObjects/Frameworks.

Michael Halliday

I haven't had any problems storing images in our database (OpenBase). We have developed many "community" based sites with photo albums as well as an online dating service, both use the same methods that Robert talked about in his message.

I have to say that we have had no performance issues, and to me this is the most elegant, scalable solution for several reasons:

- You have all of the database clustering options available to you for data redundancy & data backup.
- If your site was to grow substantially in size and you required multiple HTTP Servers (along with your multiple application servers), if your images were served by apache from the local filesystem you would run into issues with having to somehow replicate your image directories across the multiple Apache servers. I'm not saying this couldn't be done...it would just require the proper planning. Using the database for an image store would solve all of these issues.
- Using the database method, WebObjects would cache your images so you wouldn't actually have to go to the database each time.

- If you need to "associate" images with other objects I'd hate my images to be stored in the file system. You would eventually most likely run into broken links etc...this would get rather messy. Plus if you need to control access to images (I.E. Only logged on users can view product images) you would need to rely on filesystem/apache security which adds yet another layer of complexity to your application.

Again, I know many people will probably disagree with this approach. But, it is working perfectly for us and for dynamic images (or images that the user can change/upload) I think it's the most effective approach. That being said, we do use apache to serve up our static images.

I'd be interested to hear from others and there experiences with storing images in databases. You hear a lot of people saying "Don't do it, it won't perform well."...but have these people actually tried it? Or have they just been told not to do it. I have been very interested in this topic for a while now and I have done extensive searching but
never come up with any "correct" answer. I think it also depends on which database you use and how exactly the database itself stores images. I know that some are much better than others and personally this is where you'd most likely run into the performance hit (if any).

Geoff Hopson

On the Fortnum & Mason online store http://www.fortnumandmason.com, the product catalog is pretty image-heavy. Also, they (F&M) change the catalog and the associated images at least twice a year. So I wrote a tool that allows their product images to be uploaded into the database, simply for the purpose of having everything in a single place for
backup reasons. When a new catalog is ready to be deployed, the images are extracted from the database and placed under the webserver (since, as everyone notes, webservers are particularly good at vending images). The main F&M web application then gets all it's images from the webserver, as opposed to cached in the webobjects application after a fetch from the database.

However, in development, we used the images from the database directly. Command line switch toggles whether the images are read from the webserver or the database.

Doing all this means that the memory footprint is lower, since the application is not caching images, and it also means that we can do clever things with the webserver to spread the load a little.

Chuck Hill wrote something on the pros and cons of using the webserver yesterday - http://lists.apple.com/mhonarc/webobjects-dev/msg05564.html (use 'archives', 'archives' as the username/password).

Arturo Pérez

My opinion and experience FWIW, having done it both ways. I keep having this discussion so I'd thought I'd put it all down in one place.

Database pros:

- It's really easy.
- It works well.
- It's consistent with everything else; i.e. all data comes from the database.

Database cons:

- There are some tuning issues around BLOBs if you're using those (or equivalent).
- I don't understand people saying "It solves single point of failure." Didn't you just move the single point of failure from a file system to the database? Sure, databases offer replication but you can achieve the same with a clusterable file system like Transarc or the Andrew File System (AFS).
- It "makes no sense." It's just storing a bunch of non-relational data in a relational database. It feels unclean 🙂
- If you need to manipulate the data it's much much harder. For example, if it's text and you want to fix a typo it's problematic. If it's an image updating EXIF headers would be a chore. It's much easier to run a perl script over a directory of files.

Filesystem pros:

- It scales better.
- It's easier to manipulate the images if that's necessary. Particularly offline from serving them.
- It's cheaper. (Disks are cheaper costwise than databases esp. if you need a DBA).

Filesystem cons:

- It's a little harder to manage. And you might still need a database to keep track of the images.
- It's harder to build a content management system that can upload images. WebDAV might address this issue.

Well, my 2 farthings.

Chuck Hill

One largish problem with storing them in the database is that EOF will cache the data, at least for a while. For a heavily loaded site or large contents this can really chew up the memory fast.

Another alternative is to store them on the file system but not directly available to the web server. Keep an object in the database that references the data on the file system. Use Java streams to move the data from the request to the file system and from the file system into the response. This avoids the EOF overhead but allows your application to control access. It is much more efficient to have the web server directly access and vend the images etc. but if you have access restrictions this is not an option. This hybrid database / file system approach can be useful in that situation.

PetiteAbeille wrote about an EOF file system adaptor that may be of interest in relation to this question:
http://www.wodeveloper.com/omniLists/eof/2002/June/msg00053.html

Tom Pelaia

We grab images from the database in our WebObjects application (electronic logbook). It is a very heavily accessed site and allows users to make entries that have text, images and other attachments. We have found additional "pros" for loading images from a database.

Additional database pros:

- The logbook isn't the only application that uses our database. We just use a subset of the database. Other applications write to the database and make entries into our logbook and use data from our logbook. Storing complete logbook entries (images, text, attachments) in the database means other applications have easy access to logbook entries. Hence we are integrated well.
- The database is the one official source of all data
- Backups are consistent and easy. When the database is backed up, all logbook data is backed up and the backups are self consistent.
- It is easier to transfer data from one machine to another since you only have one thing to transfer.
- Since the images are provided by users, the database provides a way to avoid file names collisions. If you use a file system you need to save filenames carefully to avoid duplication.
- The first logbook we developed several years ago (without WO) stored data including images in the filesystem. That turned out to be very inflexible. If you want to reorganize the folder layout, lots of links will break. This is simply a non-issue for database storage.
- Databases are simply much more flexible. Filesystem storage is very rigid. You can store your files for the present application, but needs tend to change in time and database storage is much more flexible in supporting future needs that you might not be considering today.

Whether you choose database storage or filesystem storage really depends on your application. For our application, the electronic logbook is becoming more integrated with other systems and the database has turned out to be critical in that integration.

ocs

I happily store images in the database, but... my clients use Oracle or FrontBase. The very now though I have the misfortune to work on an application which has to use the MS-SQL thing: seems it really does not support BLOBs well (actually, the database admin just plain told me "do not use a BLOB in your tables, ever--we have the worst experience with them").

Myself, I've tried of course 🙂 (with a test database) and found that indeed there seem to be issues, like that a BLOB is never found by a WHERE clause (even if a proper value is provably given). I haven't tested for long 🙂

Thus, although I am a strong believer in storing images in the database, I can understand others who are unlucky enough not to be FrontBase users might have different opinions 🙂

The reason I am writing: before deciding where to store your images, do check the concrete database to be used. If FrontBase or Oracle, you probably would want to store them in the database, if MS-SQL, you probably would want to store them in the filesystem 🙂

Jeff

I'm a little WORusty at the moment, so please excuse any gaffes in this. WO 5.3 has renewed my interest in WebObjects.

I've experimented quite a bit with using the images in a database, and have generally found that if you do it right, there's very little performance hit to doing it compared to the file system route - Databases in general have gotten much better about handling blob data and a lot of the reasons for not storing images there simply no long apply. Plus, storing the images in the database makes writing, maintaining, and backing up your application all much easier.

Generally, I create a direct action method for dispensing the images. Something like this in your DirectAction (except that you might want to add validation if you don't want just anyone getting to your images by hacking the URL):

```
public WOActionResults imageAction()  {
  // PictureTest is an EOEntity with a BLOB containing the image data
  PictureTest pt = getPictureTestEO();
  return jpegResponseWithData(pt.image());
}

private PictureTest getPictureTestEO() {
  // Yes - you can get the session in a direct action
  //  you just need to be prepared to deal with one not existing
  // whether you return an image if no session exists depends on
 // on your own application needs.
  WOSession theSession =  existingSession();
  EOEditingContext ec = (theSession == null) ?  new EOEditingContext() : theSession.defaultEditingContext();
  String picid = (String)request().formValueForKey("picid");
  return (PictureTest)EOUtilities.objectMatchingKeyAndValue(ec, "PictureTest","id", new Integer(picid));
}

private WOResponse jpegResponseWithData(NSData theData) {
 // This method returns the data so that the browser
 // recognizes the image type. In this particular application
 // I've just hardcoded a mime type of JPEG because I only
 // use JPEG images, but a better way would be to store the mime-type
```

```
 // that corresponds to the image data in the BLOB as a separate
 // field. I might revise this sample later on to show that.
  WOResponse response = WOApplication.application().createResponseInContext(context());
  response.appendHeader("image/jpeg", "Content-Type");
  response.appendContentData(theData);
  return response;
}
```

Then, in your WOComponent, you create a virtual accessor like this:

```
public String imageURL() {
  return context().directActionURLForActionNamed("icon", null) + "?picid=" + pictureItem.id();
}
```

in this case, pictureItem is a PictureItem EOEntity instance that I use in a WORepetition - I'm just pulling the id number from the currently selected picture.

Then, in WOBuilder, you bind the WOImage or WOActiveImage's src binding to imageURL

This approach eliminated a lot of the overhead that you get by just binding directly to the EOEntity's attribute, and really isn't much extra work.

Bill Bumgarner

Storing images in the database is generally a bad idea for a whole slew of reasons. First and foremost, it is loads slower than serving images directly from the web server and it completely bypasses numerous automatic "optmiziations" that are present when serving from a filesystem. If it can't be avoided, it can't be avoided.... however, if you have any hopes of scaling your solution to a large community of users or a heavy hit rate, expect to expend a lot of engineering and hardware dollars making images-in-the-database go fast.

In particular:

Images are normally served statically from the filesystem... because you are now serving them as dynamic content, the following performance hits occur:

- no client side caching ouch five copies of a single image on a page yields five seperate hits on WO.
- image requests must be serialized-- not only do IMAGE hits have to be serialized, but all other hits on the WOF app will have to wait for any pending image hits to be handled. In terms of Netscape's REALLY SLOW table layout algorithm that requires the size of all images to be known, this means that the user WON'T see the contents of the table until ALL image hits have returned at least the size of the image.... since hits are serialized, that means that all but the last image must be entirely handled.
- performance difference between a static hit vs. a fully dynamic hit is tremendous in favor of static. Think about it... a static hit basically means the web server opens a file, reads/writes the contents to a socket, closes... dynamic hits require IPC, a database round trip maybe, a bunch of memory munging, a pass through request/response, etc.etc.etc...
- no server side caching; every instance of your app will end up with a copy of every image served in its memory. As well, the IPC between database and WO app server will have to pass all that data back and forth, as well.
- most databases are not designed to handle BLOBs well.... regardless

Alternatives?

Stick a path in the filesystem in your database instead of a blob; abstract arbitrarily to facilitate administration, etc...
If you REALLY need the images to come from the database, build an image manager that maintains a hierarchy of the images in the filesystem and arbitrates the updates between the database and the images.

One thought; if an image needs to be refreshed and you are worried about client-side or proxying-firewall caching, rename the image in the filesystem (or move it) and generate a new URL-- this should be the image managers responsibility.