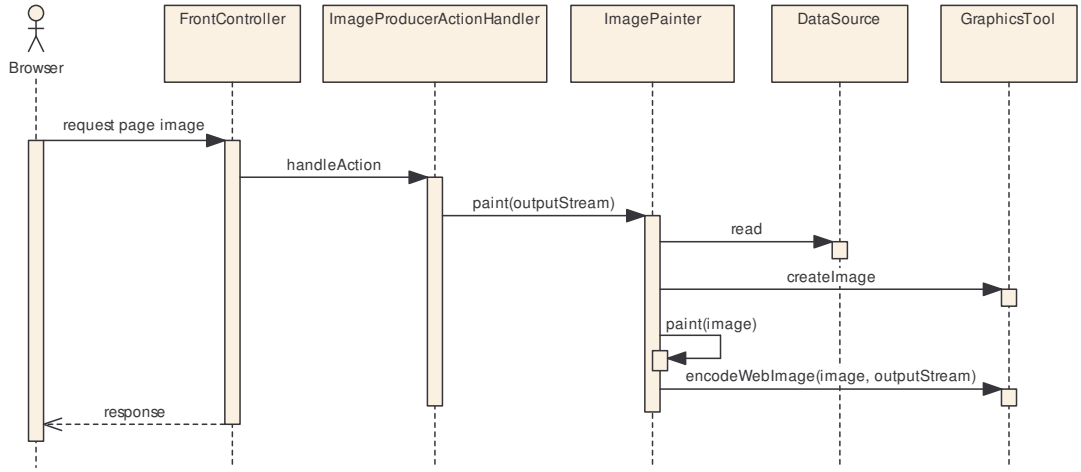
Image Producer

Generates a graphic image on the fly that is compatible with Web browsers. Allows the business to supply to consumers graphs built from live, up-to-date business data.



Background

The Web site consumer requests a graph that visually depicts the state of a dataset of interest. The Web component container dispatches the request to a custom component. The custom component reads the consumer's selected dataset and produces an image that allows the consumer to examine a graphical view of the dataset.

Value and Benefits

It's relatively easy to enlist a graphic artist to produce a static graphic image that can be placed in a Web page. As you know, however, the static graphic only reflects an artist's rendition of some context at the time it was designed. If the static graphic represented a data context, then the image may be obsolete by the time it is published to the Web site and/or viewed by consumers. We obviously cannot use this approach when we are reflecting visuals of up-to-date, "live" data. Instead the graphic image must be produced on the fly.

Web browsers are not able to display every graphic image format. If you produce an image that is native to the toolkit you are using to generate the graph on the fly, the raster or vector will likely be incompatible with what the consumer's browser can display. Therefore, the graphic image must also be converted to a browser-compatible format before embedding it in the response. What is more, the converted image must be written into the response buffer that the browser can read.

The *Image Producer* pattern defines the necessary steps to accomplishing these tasks. The benefits to the consumer include being able to view their selected dataset of interest graphically, with up-to-the-second accuracy. The business benefits by adding value to their information systems managed by their *Dynamic Web Site (page #)*.

Putting It to Work

The consumer makes a request, and the Web site must produce a response using *Dynamic Web Page (page #)*. The page response must include a reference to the generated image so the browser can request it. The image does not yet exist, however. Therefore, the generated URL in the page response must refer to a Web custom component that knows how to produce the image dynamically. The URL is included in a standard HTML IMG tag:

```
<IMG SRC="/webAppName/ImageProducerServlet?featureId=123"/>
```

When the browser parses the page it will encounter the IMG tag. The browser will then send another request to the Web site asking for the image by its source URL (`SRC="..."`). The URL does not reference a static image, but, rather, a custom component (`ImageProducerServlet`). The Web server dispatches the request to the custom component. The custom component gets any pertinent parameter(s) (`featureId=123`) and uses the parameter(s) to determine what kind of image it needs to produce.

Next the custom component gathers the data from the consumer-specified data source(s). It then creates a graphic image using its native graphics toolkit and executes an image production strategy using the gathered live data. Once the native image is produced it must convert the image to one that is compatible with the Web browser. This typically means `.GIF` or `.JPG`.

When the browser-compatible image is produced it is placed into the response output stream. You must indicate the content type in the response header. Depending on the image type you produce, you will use one of the following to content types:

```
image/jpeg
image/gif
```

You must not use a “cooked” output stream when outputting the image bytes. Use a raw file, one that accepts bytes that are in no way interpreted as they are written to the output stream.

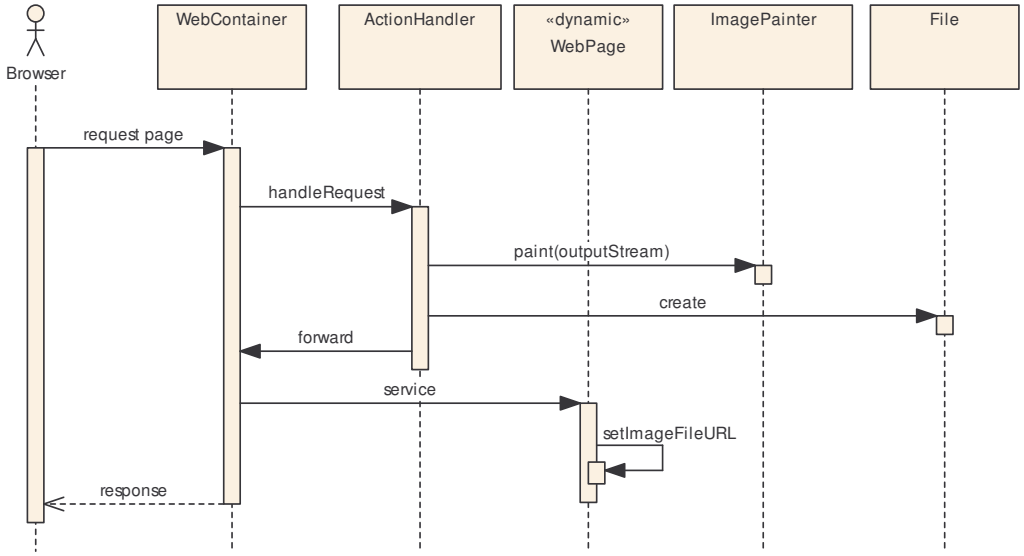
Producing Disk Image Files

You will have to determine whether or not to store images on disk. I recommend using the above procedure, not generating disk files. However you could take the approach of generating the image and saving it to a disk file when the page is first requested (rather than generating it later). You would in that case generate a URL to the static image on disk. The browser would then request the generated static image file, for example:

```
<IMG SRC="/images/tmp/type123_035722193.tmp.gif"/>
```

There are several problems with this approach: How do we clean up after ourselves and remove the temporary image files when they are no longer needed? We may conclude that there is an advantage to leaving the file on disk since it could be reused on subsequent requests (see *Server-Side Image Caching* below). Granted, this may help to lessen the load on the server-side application. But how would we determine when each image is obsolete and that it could be deleted? How would you arrange for the “obsolete image file strategy” to be executed, and how would it be implemented? How could you

be certain that you would have enough disk space to store all the images generated in behalf of all users during any given peak usage period?



Further, there may be issues around creating files that can be referenced statically by the Web server. If your Web server and your Web component container are two separate processes, and the probably are, how will your Web custom component save an image file into a directory on the Web server that is mapped to your specific application's site root?

With all of the potential problems that must be addressed, you have probably concluded as I did, that providing just-in-time images is a better choice. But you may decide to use this approach for caching purposes, if caching provides advantages in your business domain.

Server-Side Image Caching

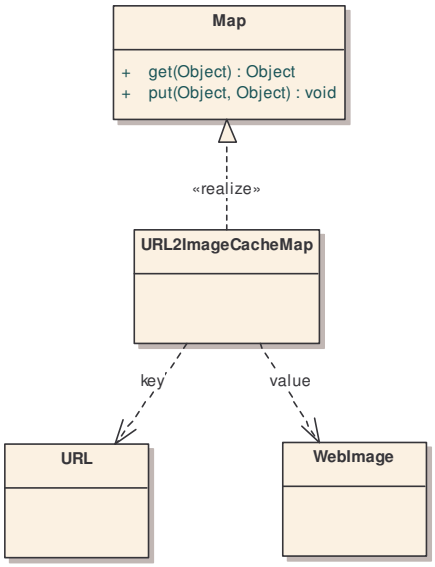
Is image caching a worthwhile effort? If so, how should it be accomplished?

As previously stated, caching reusable dynamically produced images in a temporary directory could help to lessen the load on the server-side application. But is caching really necessary in your business domain? If so, consider some of the following image caching strategies and their caveats.

The requested URL could be used to determine whether a consumer is requesting an image previously requested. If a URL is identical to one previously received, the previously generated image could be reused. In that case we would probably create a URL-to-image-file mapping. This approach would have some overhead since you may choose to store the mapping in a persistent store. Otherwise, if the server is stopped you may as well remove all disk cached image files and start from scratch upon server restart. So calculate the value of preserving a mapping and searching it on each dynamic image request.

You could name the image file based on the URL that created it. This would negate the need to create a URL-to-image-file mapping in a database. But you would still have to hit the disk on each request to find out whether or not the image already exists.

We could cache images in memory inside the Web component container. We might accomplish this by creating a URL-to-image mapping using a `Dictionary` or `Map` object. We would use the URL string as the key for retrieving the image object. This approach would only keep images cached while the server is running, unless we persist and re-read the mapping whenever the server is bounced. But there is potentially a huge memory overhead to this approach. Similar to the disk caching approach, how could we be certain that there would be enough memory to store all the images generated in behalf of all users during any given peak usage period?



Reaping memory back from the in-memory strategy may be an issue. We could use a background thread to destroy cached images if they have not been used by the end of some predetermined time period. Caution should be used with this approach, however. Starting background threads may not be legal on a given platform. For example, starting a background thread might be permitted on the Web tier, but not on the business tier. The EJB specification stipulates that user-created threads inside the EJB container are forbidden. There are ways to get around the limitation, but keep this in mind if you decide to implement a memory reaper for in-memory caching.

So how would caching benefit your enterprise? A dynamically produced image that is requested multiple times using identical data may be rare in many business domains. You will have to determine if this is the case in your specific domain. In many cases identical requests are more likely to occur for the same consumer. If that is true in your domain, remember that the consumer's Web browser will probably cache the image on the client side. Subsequent requests for the identical URL will more than likely cause the browser to reload the image from its own client-side cache. Hence, server-side caching may actually be wasteful.

With that thought, a related concern now comes to the fore. What if we want to guarantee that the browser never recalls a cached dynamically produced image, and that a fresh image is always generated? Make sure that each *Dynamic Web Page (page #)* response somehow generates a unique URL every time for each user. Here is one strategy to accomplish this:

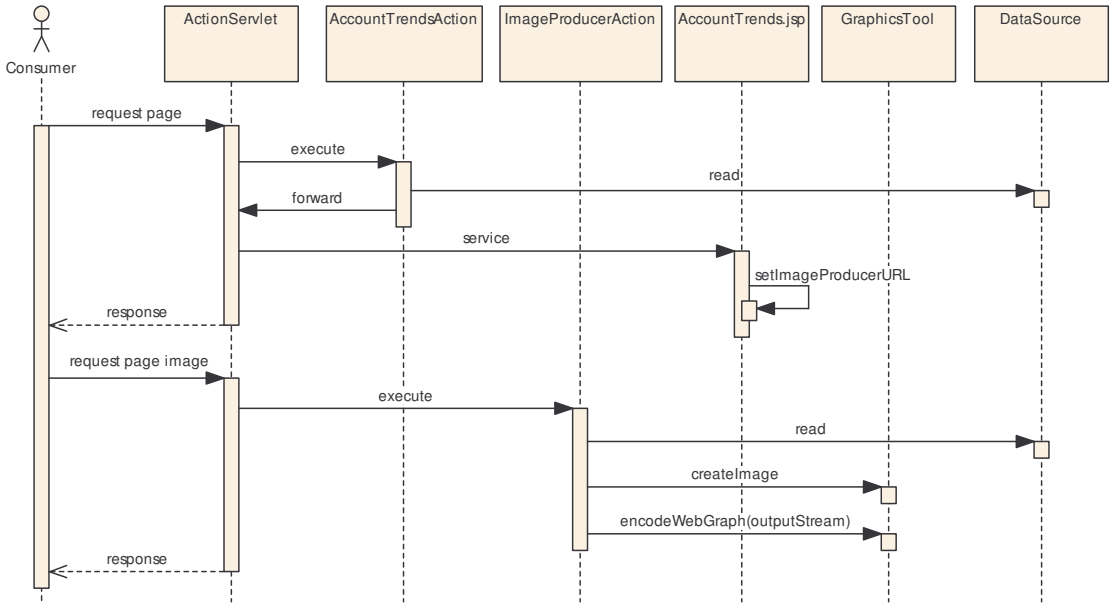
```
<% long tempNow = System.currentTimeMillis() %>
. . .
<IMG SRC="/webAppName/ImageProducerServlet?featureId=123&uniqueId=<%= tempNow %>"/>
```

This URL adds an additional parameter, `uniqueId`. The additional parameter is set to the system time in milliseconds. Since it is virtually impossible for the same consumer to make two or more requests at exactly the same time, the above URL can be considered absolutely unique. There is no reason for the server-side component to make use of the `uniqueId` parameter when it generates the image. In fact, it will probably always be ignored. This unique identifier simply ensures that the browser will consider every dynamic image request to be unique, which will ensure that the image will always be requested anew.

Example

The following sequence diagram illustrates the behavior of the implementation strategies I use in my example. I do not attempt the use of an image caching mechanism. The thrust of my example is to demonstrate the core concepts behind a Java-based *Image Producer*.

The sequence diagram is more of an analysis-level diagram rather than design-level. It is meant to show the logical intentions of the solution, not the details. However the subsequent code snippets provide a fair amount of detail.



I have based my example on the use of the *Struts* framework, but my intention is to highlight *Image Producer*, not *Struts*. The example I present is the happy-path flow in a *Show Account Trends* use case. The requirement is to show consumers graphically what activities have occurred in their stock trading account over a selected period of time.

The *Consumer* requests a specific page from our Web server. The *Struts* `ActionServlet` gets control from the servlet container (perhaps *Tomcat*). It then dispatches the request to an action handler object of the class `AccountTrendsAction`. The `AccountTrendsAction` object retrieves some data

from a data source and prepares it to be forwarded to the page `AccountsTrends.jsp`. Now as we peer inside that *Dynamic Web Page (page #)* we find the following HTML and scriptlet code:

```
<IMG SRC="/traderApp/ImageProducer.do?img=acctTrends&accountRef=<%=
trader.getAccountRef() %>"/>
```

This produces markup to the page output stream that the consumer's Web browser encounters while parsing the page. The actual markup text may end up looking something like the following:

```
<IMG SRC="/traderApp/ImageProducer.do?img=acctTrends&accountRef=A72547731ZZY"/>
```

When the browser sees this `IMG` tag, it requests the URL found in the value of the tag's `SRC` attribute from the Web server:

```
/traderApp/ImageProducer.do?img=acctTrends&accountRef=A72547731ZZY
```

When the `ActionServlet` receives this request (by way of the Web server and servlet container), it dispatches action handling to the handler mapped to `ImageProducer.do` in `struts-config.xml`. In this case it is an instance of class `ImageProducerAction`. Inside class `ImageProducerAction` we find the code:

```
public ActionForward execute(
    ActionMapping anActionMapping,
    ActionForm anActionForm,
    HttpServletRequest aRequest,
    HttpServletResponse aResponse) throws Exception
{
    String tempAccountRef = anActionForm.getAccountRef();
    AccountTrendInfo tempTrendInfo = this.getAccountTrendInfo(tempAccountRef);
    this.produceImage(anActionForm, aResponse, tempTrendInfo);
    return null; // response complete; back to browser
}

protected AccountTrendInfo getAccountTrendInfo(String anAccountRef)
{
    // . . .
}

protected Dimension getImageSize(ActionForm anActionForm)
{
    // . . .
}

protected void paint(Graphics aGraphics, AccountTrendInfo anAccountTrendInfo)
{
    // . . .
}

protected void produceImage(
    ActionForm anActionForm,
    HttpServletResponse aResponse,
    AccountTrendInfo anAccountTrendInfo) throws Exception
{
    Dimension tempDim = this.getImageSize(anActionForm);
    Frame tempFrm = new Frame();
    tempFrm.addNotify();
    Image tempImage = tempFrm.createImage(tempDim.width, tempDim.height);
    this.paint(tempImage.getGraphics(),anAccountTrendInfo);
    aResponse.setContentType("image/gif");
    OutputStream tempOutputStream = aResponse.getOutputStream();
```

```
    GifEncoder tempGif = new GifEncoder(tempImage, tempOutputStream);
    tempGif.encode();
    tempOutputStream.flush(); // to browser
}
```

The action handler's `execute()` method uses the `accountRef` URL parameter to retrieve data from the data store containing consumer accounts (details of `getAccountTrendInfo()` not shown). Then the real fun begins. The method `produceImage()` is invoked. It might be that the URL parameters include some indication of the image size to be produced. The internal `getImageSize()` method determines the existence of such parameters, and if missing answers a default size. With the image size specified, the `Image` object is now created. The internal `paint()` method is used to paint the trends graph on the `Image` (again details not shown). Finally the `HttpServletResponse` object is prepared for output. The `image/gif` content type is set. Then the graphics image is put on the response object's raw output stream by the `GifEncoder` tool (see *Frameworks and Tools* below). The final step of flushing the output stream is important as it causes the output stream to be sent to the requesting browser. Also, answering null from the `execute()` method is necessary to tell the *Struts* framework that the response is complete, and not to forward.

I suggest that you create a simple abstract class that implements the reusable behavior of this example. The `GifServlet` (see *Frameworks and Tools* below) provides such a framework for the *Acme* `GifEncoder`. This limits you to implementing *Image Producer* through a servlet. If you are using Struts you will want to create an abstract `Action` subclass, or better yet, make a plain-old-Java-object implementation that is reusable with all Web or business platform and frameworks.

Consequences

Tradeoffs exist among the following competing forces within the *Image Producer* solution pattern.

- ***Generate Image Files If Advantageous***: There are several complexities around this approach, including accessing the appropriate directory and housekeeping. You will want to use this strategy if you determine that image file caching is useful in your environment.
- ***Use Caching If Necessary***: Server-side image caching may reduce processing load, but it will be more complex to implement properly than to function without it. Determine whether or not caching will provide an advantage in your business domain before going to the trouble of implementing it.
- ***Select the Right Tier***: Consider the impact of dynamic image production on each tier. Both load and required facilities should be calculated. If throughput will be poor or necessary facilities are lacking, place the *Image Producer* on a different tier. If your business must produce a very high volume of dynamic images consider dedicating a separate tier to your *Image Producer*. This will be more costly in hardware, software, development, and administration. Nonetheless, if you business requires high volume image throughput a dedicated tier may be a necessity.

Frameworks and Tools

- **Java Tools**: If you are using Java, the *JDK* includes the class `JPEGImageEncoder` that is used to convert an *AWT* image into a *JPEG* format. If you prefer to generate *GIF* images in Java, you should look at the *Acme* (really!) *JMP* package, and the `GifEncoder` class in particular. You may also find *Sun Microsystem's JIMI* product useful. Further, *Aaron Porter* has written a `GifServlet` for J2EE dynamic image production.

Image Producer: 8
Date/Time: 9/7/2004 1:18 PM