Caches

WebHare offers various caches to speed up page serving or prepare and reduce image sizes

Image cache

You can generate URLs to images stored in WRD, the Publisher and Publisher metadata.

Image Resizing Methods

For use with image resizing functions like GfxResizeImageBlobWithMethod.

The following properties can be set in the resizing method:

- *method*: how the image should be resized
- *setwidth*, *setheight*: the new image dimensions, may not be the actual resulting image dimensions. when a method doesn't require either of these, you can set it to 0 or leave the cell out of the method. (ie [method := "scale", setwidth := 640, setheight := 0] and [method := "scale", setwidth := 640] are equivalent)
- *format*: output image MIME type, one of "image/jpeg", "image/gif" or "image/png". If left empty, the output image will have the same type as the input image if possible; "image/x-bmp" is converted to "image/png" and "image/tiff" is converted to "image/jpeg".
- *bgcolor*: the background color to apply when working with (semi-)transparent images or the 'canvas' resizing functions
- *quality*: if the output image has the "image/jpeg" type, the JPEG quality to use (a value between 0-100)
- *fixorientation*: try to fysically rotate the image according to the image rotation stored within the image (defaults to true)
- *noforce*: return the input image if the chosen method and parameters didn't actually change the image (defaults to true)
- *grayscale*: discard color information in the image (30% red, 59% green, 11% blue)

The *method* cell is always required, other cells may be required depending on the method.

The image cache will not touch images that already have the correct dimensions and rotation unless the *noforce* parameter is set. This is a route that can be used to take exact control of the quality and to permit the use of animated GIFs - if the end user does the work to properly resize the image, WebHare won't touch it.

Image methods

To illustrate the different methods, the following reference image is used:

 (225x150)

none

The image is left untouched and returned as-is.

fit

If either the width of the image is larger than *setwidth* or the height is larger than *setheight*, the image is scaled down proportionally to fit within *setwidth* and *setheight*. If only one of *setwidth* or *setheight* is given, only the given direction is checked. This only make the image smaller if it's too big.

[method := "fit", setwidth := 100, setheight := 100]
 (100x67)

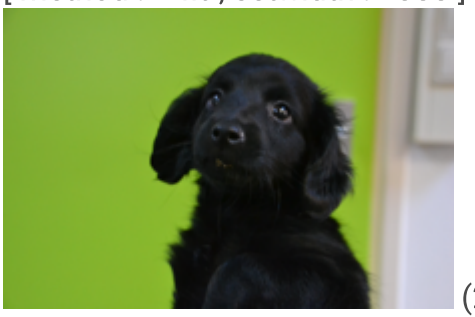
[method := "fit", setwidth := 300, setheight := 300]
 (225x150)

[method := "fit", setwidth := 100]
 (100x67)

[method := "fit", setheight := 100]

 (150x100)

[method := "fit", setwidth := 300]

 (225x150)

[method := "fit", setheight := 300]

 (225x150)

scale

The same as *fit*, but if the image is also scaled up proportionally if it's smaller than the given *setwidth* or *setheight*. Equivalent to CSS: "background-size: contain;".

[method := "scale", setwidth := 100, setheight := 100]

 (100x67)

[method := "scale", setwidth := 300, setheight := 300]

 (300x200)

[method := "scale", setwidth := 100]

 (100x67)

[method := "scale", setheight := 100]

 (150x100)

[method := "scale", setwidth := 300]

 (300x200)

[method := "scale", setheight := 300]

 (450x300)

fitcanvas, scalecanvas

The same as *fit* or *scale* but if both *setwidth* and *setheight* are given, the resulting image will always be *setwidth* by *setheight* pixels. Any portion of the resulting canvas not covered by the orignal (scaled) image is filled with *bgcolor*.

[method := "fitcanvas", setwidth := 100, setheight := 100, bgcolor := "red"]

 (100x100)

[method := "fitcanvas", setwidth := 300, setheight := 300, bgcolor := "red"]

 (300x300)

[method := "scalecanvas", setwidth := 100, setheight := 100, bgcolor := "red"]

 (100x100)

[method := "scalecanvas", setwidth := 300, setheight := 300, bgcolor := "red"]

 (300x300)

fill

The image is scaled proportionally until it covers the *setwidth* by *setheight* area. If only one of *setwidth* and *setheight* is given, it behaves like the *scale* method, otherwise equivalent to CSS: "background-size: cover;".

[method := "fill", setwidth := 100, setheight := 100]

 (100x100)

[method := "fill", setwidth := 300, setheight := 300]

 (300x300)

Legacy resize methods

The old methods 'stretch', 'stretch-x', 'stretch-y', 'crop' and 'cropcanvas' are rarely used nowadays and are error-prone when used. These methods have been deprecated and will not be supported in TypeScript or for WEBP/AVIF outputs

Picking a method:

- Should the image fill the target, discrding parts if necessary? 'fill'
- Should the image be scaled up if needed? ‘scale’, otherwise ‘fit’.
- And should the image be on a fixed size canvas, with padding/letterboxing? Use 'scalecanvas' and 'fitcanvas' instead of 'scale' and 'fit'

You can use the webdesign siteconfig cache to hold data that doesn't generally change from page to page - eg the structure of a site menu (except for any 'selected' cell). You could build this yourself using the adhoc cache, but it may be hard to get all the eventmasks for invalidations right.

The webdesign->siteconfig returns cached configuration that will be originally provided by GetcacheableSiteConfig. Specifically, this cache will be invalided by

- any changes to the webdesign source code (or its dependencies)
- any changes to data in the site (file sor folders)
- updates to any moduledefinition or site profile

And the cached data is stored per site (ie, all files and folders in the same site will share the cached siteconfig)

Override the GetCacheableSiteConfig function to return the data that doesn't change per site, for example:

The GetCacheableSiteConfig function may return a cell named eventmasks containg a string array. These will be added to the list of eventmasks associated with this key.

Ad-Hoc cache

The ad-hoc cache is used to cache frequently regenerated HareScript data. It works by invoking GetAdhocCached with a record as a key and a callback function to generate the data. The cache will see if it has the data for the specified key, and if not, will invoke the callback to generate the data.

The key is combined with the calling library to create a hash for the cache key. This prevents adhoc cache calls made from different libraries from having conflicting keys but will also generally require you to use the same route (or at least the same library) to an adhoc cache call.

When multiple jobs in the same WebHare process request the same key they will all wait for one job to actually do the calculation, and then all share the result.

Example

```
<?wh
LOADLIB "wh::datetime.whlib";
LOADLIB "module::system/cache.whlib";

RECORD FUNCTION GetCachedData()
{
  RETURN [ value := [[ a := GetCurrentDateTime() ]]
         , ttl := 15 * 60000 // 15 mins
         ];
}

RECORD ARRAY items := GetAdhocCached([ type := "data" ], PTR GetCachedData());
```

Output

```
+RECORD ARRAY
 +RECORD
  +A: 2014-10-13 (286) 15:22:01.559'
```

Caveats

- The adhoc cache is synchronized between all HareScript jobs in the same process and thus not 'free' to call! Overusing the adhoc cache may actually slow down your application - profiling is recommended.
- The adhoc cache key must match exactly for data to match. You should be especially careful when passing date/times to the adhoc cache derived from GetCurrentDatetime - if necessary, round them first using GetRoundedDatetime.
- The adhoc cache is shared at the WebHare process level:
  - All dynamic pages run inside the webserver and share their cache
  - Managed tasks and publications may or may not share a cache
  - Runscript (wh run) tasks will not share their cache between runs.
- Caching blobs that are selected from the database will cause them to be downloaded instead of just a reference being held. An indication of this is usually StoreAdhocCached appearing high in your performance profiles. If your cache stores images or files, consider converting them first to URLs using the image cache.