# Nginx Haskell module
## (yet another doc with examples)

Alexey Radkov

January 25, 2019

# Contents

# Why bother?

The *nginx-haskell-module* allows for running in Nginx written in Haskell synchronous and asynchronous tasks, request body handlers, per-worker and shared services, and content handlers.

# Synchronous tasks

Synchronous tasks are mostly *pure* Haskell functions of various types. To make them available in Nginx configuration files, they must be exported with special declarations named *exporters*. Below is a table of *type/exporter* correspondence for all available synchronous handlers.

| Type | Exporter |
| --- | --- |
| String -> String | ngxExportSS (NGX_EXPORT_S_S) |
| String -> String -> String | ngxExportSSS (NGX_EXPORT_S_SS) |
| String -> Bool | ngxExportBS (NGX_EXPORT_B_S) |
| String -> String -> Bool | ngxExportBSS (NGX_EXPORT_B_SS) |
| [String] -> String | ngxExportSLS (NGX_EXPORT_S_LS) |
| [String] -> Bool | ngxExportBLS (NGX_EXPORT_B_LS) |
| ByteString -> L.ByteString | ngxExportYY (NGX_EXPORT_Y_Y) |
| ByteString -> Bool | ngxExportBY (NGX_EXPORT_B_Y) |
| ByteString -> IO L.ByteString | ngxExportIOYY (NGX_EXPORT_IOY_Y) |

All synchronous handlers may accept *strings* (one or two), a *list of strings*, or a *strict bytestring*, and return a *string*, a *boolean* or a *lazy bytestring*. The last handler from the table is *impure* or *effectful*, and it returns a *lazy bytestring* wrapped in *IO Monad*.

There are two kinds of exporters which differ only in their implementations. The first kind — *camel-cased* exporters — is implemented by means of *Template Haskell*, the other kind — exporters in braces, as they are shown in the table — is implemented using *CPP macros*. Both of them provide *FFI* declarations for functions they export, but the camel-cased exporters are available only from a separate Haskell module *ngx-export*, which can be downloaded and installed by *cabal*, whereas the CPP exporters are implemented inside the *nginx-haskell-module* in so-called *standalone* approach, where custom Haskell declarations get wrapped inside common Haskell code.

## Examples

In all examples in this section and later we will use *modular* approach with *camel-cased* exporters and separate compilation of Haskell code.

**File test.hs**

```haskell
{-# LANGUAGE TemplateHaskell #-}

module NgxHaskellUserRuntime where

import           NgxExport
import qualified Data.Char as C

toUpper :: String -> String
toUpper = map C.toUpper
ngxExportSS 'toUpper

ngxExportSS 'reverse

isInList :: [String] -> Bool
isInList [] = False
isInList (x : xs) = x 'elem' xs
ngxExportBLS 'isInList
```

In this module we declared three synchronous handlers: *toUpper*, *reverse*, and *isInList*. Handler *reverse* exports existing and well-known Haskell function *reverse* which reverses lists. Let's compile *test.hs* and move the library to a directory, from where we will load this.

```
ghc -O2 -dynamic -shared -fPIC -L$(ghc --print-libdir)/rts -lHSrts-ghc$(↙
    ↳ ghc --numeric-version) test.hs -o test.so
[1 of 1] Compiling NgxHaskellUserRuntime ( test.hs, test.o )
Linking test.so ...
cp test.so /var/lib/nginx/
```

**File test.conf**

```nginx
user                nginx;
worker_processes    4;

events {
    worker_connections  1024;
}

http {
    default_type        application/octet-stream;
    sendfile            on;

    haskell load /var/lib/nginx/test.so;

    server {
        listen          8010;
        server_name     main;

        location / {
            haskell_run toUpper $hs_upper $arg_u;
```

```
            haskell_run reverse $hs_reverse $arg_r;
            haskell_run isInList $hs_isInList $arg_a $arg_b $arg_c $arg_d;
            echo "toUpper $arg_u = $hs_upper";
            echo "reverse $arg_r = $hs_reverse";
            echo "$arg_a 'isInList' [$arg_b, $arg_c, $arg_d] = $hs_isInList";
        }
    }
}
```

Library *test.so* gets loaded by Nginx directive *haskell load*. All synchronous handlers run from directive *haskell_run*. The first argument of the directive is a name of a Haskell handler exported from the loaded library *test.so*, the second argument is an Nginx variable where the handler will put the result of its computation, the rest arguments are passed to the Haskell handler as parameters. Directive *haskell_run* has *lazy* semantics in the sense that it runs its handler only when the result is needed in a content handler or rewrite directives.

Let's test the configuration with *curl*.

```
curl 'http://127.0.0.1:8010/?u=hello&r=world&a=1&b=10&c=1'
toUpper hello = HELLO
reverse world = dlrow
1 'isInList' [10, 1, ] = 1
```

# Synchronous content handlers

There are three types of exporters for synchronous content handlers.

| Type | Exporter |
| --- | --- |
| ByteString -> ContentHandlerResult | ngxExportHandler (NGX_EXPORT_HANDLER) |
| ByteString -> L.ByteString | ngxExportDefHandler (NGX_EXPORT_DEF_HANDLER) |
| ByteString -> UnsafeContentHandlerResult | ngxExportUnsafeHandler (NGX_EXPORT_UNSAFE_HANDLER) |

Types *ContentHandlerResult* and *UnsafeContentHandlerResult* are declared as type synonyms in module *NgxExport*.

```
type ContentHandlerResult = (L.ByteString, ByteString, Int)
type UnsafeContentHandlerResult = (ByteString, ByteString, Int)
```

All content handlers are *pure* Haskell functions, as well as the most of other synchronous handlers. The *normal* content handler returns a *3-tuple (response-body, content-type, HTTP-status)*. The response body consists of a number of chunks packed in a *lazy bytestring*, the content type is a *strict bytestring* such as *text/html*. The *default* handler defaults the content

type to *text/plain* and the HTTP status to *200*, thus returning only chunks of the response body. The *unsafe* handler returns a *3-tuple* with a single-chunked response body, the content type and the status, but the both bytestring parameters are supposed to be taken from static data, which must not be cleaned up after request termination.

*Normal* and *default* content handlers can be declared with two directives: *haskell_content* and *haskell_static_content*. The second directive runs its handler only once, when the first request comes, and returns the same response on further requests. The *unsafe* handler is declared with directive *haskell_unsafe_content*.

## An example

Let's replace Nginx directive *echo* with our own default content handler *echo*. Add in *test.hs*,

```haskell
import           Data.ByteString (ByteString)
import qualified Data.ByteString.Lazy as L

-- ...

echo :: ByteString -> L.ByteString
echo = L.fromStrict
ngxExportDefHandler 'echo
```

compile it and put *test.so* into */var/lib/nginx/*. Add new location */ch* into *test.conf*,

```
        location /ch {
            haskell_run toUpper $hs_upper $arg_u;
            haskell_run reverse $hs_reverse $arg_r;
            haskell_run isInList $hs_isInList $arg_a $arg_b $arg_c $arg_d;
            haskell_content echo
"toUpper $arg_u = $hs_upper
reverse $arg_r = $hs_reverse
$arg_a `isInList` [$arg_b, $arg_c, $arg_d] = $hs_isInList
";
        }
```

and test again.

```
   curl 'http://127.0.0.1:8010/ch?u=content&r=handler&a=needle&b=needle&c=in↙
      ↳ &d=stack'
 toUpper  content = CONTENT
 reverse  handler = reldnah
 needle `isInList` [needle, in, stack] = 1
```

## Asynchronous tasks and request body handlers

There are two types of Haskell handlers for per-request asynchronous tasks: an asynchronous handler and an asynchronous request body handler.

| Type | Exporter |
|---|---|
| `ByteString -> IO L.ByteString` | `ngxExportAsyncIOYY` `(NGX_EXPORT_ASYNC_IOY_Y)` |
| `L.ByteString -> ByteString -> IO L.ByteString` | `ngxExportAsyncOnReqBody` `(NGX_EXPORT_ASYNC_ON_REQ_BODY)` |

Normal asynchronous handler accepts a strict bytestring and returns a lazy bytestring. Its type exactly corresponds to that of the handlers exported with *ngxExportIOYY*. Request body handler additionally accepts request body chunks in its first parameter.

Unlike synchronous handlers, asynchronous per-request handlers are *eager*. This means that they will always run when declared in a location, no matter whether their results are going to be used in the response and rewrite directives, or not. The asynchronous handlers run in an early *rewrite phase* (before rewrite directives), and in a late rewrite phase (after rewrite directives, if in the final location there are more asynchronous tasks declared). It is possible to declare many asynchronous tasks in a single location: in this case they are spawned one by one in order of their declarations, which lets using results of early tasks in inputs of later tasks. This ordering rule extends naturally beyond hierarchical levels: tasks declared in *server* clause run before tasks from *location* clauses, while tasks from *location-if* clauses run latest.

Asynchronous tasks are bound to the Nginx event loop by means of *eventfd* (or POSIX *pipes* if eventfd was not available on the platform when Nginx was being compiled). When the rewrite phase handler of this module spawns an asynchronous task, it opens an eventfd, then registers it in the event loop, and passes it to the Haskell handler. As soon as the Haskell handler finishes the task and pokes the result into buffers, it writes into the eventfd, thus informing the Nginx part that the task has finished. Then Nginx gets back to the module's rewrite phase handler, and it spawns the next asynchronous task, or returns (when there are no more tasks left), moving request processing to the next stage.

## An example

Let's add two asynchronous handlers into *test.hs*: one for extracting a field from POST data, and the other for delaying response for a given number of seconds.

**File test.hs** (*additions*)

```haskell
import qualified Data.ByteString.Char8 as C8
import qualified Data.ByteString.Lazy.Char8 as C8L
import           Control.Concurrent
import           Safe


-- ...


reqFld :: L.ByteString -> ByteString -> IO L.ByteString
reqFld a fld = return $ maybe C8L.empty C8L.tail $
    lookup (C8L.fromStrict fld) $ map (C8L.break (== '=')) $ C8L.split '&' a
ngxExportAsyncOnReqBody 'reqFld


delay :: ByteString -> IO L.ByteString
```

```
delay v = do
    let t = readDef 0 $ C8.unpack v
    threadDelay $ t * 1000000
    return $ C8L.pack $ show t
ngxExportAsyncIOYY 'delay
```

This code must be linked with *threaded* Haskell RTS this time!

```
ghc -O2 -dynamic -shared -fPIC -L$(ghc --print-libdir)/rts ↙
    ↳ -lHSrts_thr-ghc$(ghc --numeric-version) test.hs -o test.so
[1 of 1] Compiling NgxHaskellUserRuntime ( test.hs, test.o )
Linking test.so ...
cp test.so /var/lib/nginx/
```

Let's make location */timer*, where we will read how many seconds to wait in POST field *timer*, and then wait them until returning the response.

**File test.conf** (*additions*)

```
location /timer {
    haskell_run_async_on_request_body reqFld $hs_timeout timer;
    haskell_run_async delay $hs_waited $hs_timeout;
    echo "Waited $hs_waited sec";
}
```

Run curl tests.

```
curl -d 'timer=3' 'http://127.0.0.1:8010/timer'
Waited 3 sec
curl -d 'timer=bad' 'http://127.0.0.1:8010/timer'
Waited 0 sec
```

# Asynchronous content handlers

There are two types of *impure* content handlers that allow for effectful code. One of them corresponds to that of the *normal* content handler, except the result is wrapped in *IO Monad*. The other accepts request body chunks in its first argument like the handler exported with *ngxExportAsyncOnReqBody*.

| Type | Exporter |
| --- | --- |
| ByteString -> IO ContentHandlerResult | ngxExportAsyncHandler (NGX_EXPORT_ASYNC_HANDLER) |
| L.ByteString -> ByteString -> IO ContentHandlerResult | ngxExportAsyncHandlerOnReqBody (NGX_EXPORT_ASYNC_HANDLER_ON_REQ_BODY) |

The first handler is declared with directive *haskell_async_content*, the handler that accepts

request body chunks is declared with directive *haskell_async_content_on_request_body*.

It's easy to emulate effects in a synchronous content handler by combining the latter with an asynchronous task like in the following example.

```
location /async_content {
    haskell_run_async getUrl $hs_async_httpbin "http://httpbin.org";
    haskell_content echo $hs_async_httpbin;
}
```

Here *getUrl* is an asynchronous Haskell handler that returns content of an HTTP page. This approach has at least two deficiencies related to performance and memory usage. The content may be huge and chunked, and its chunks could be naturally reused in the content handler. But they won't, because here they get collected by directive *haskell_run_async* into a single chunk, and then passed to the content handler *echo*. The other problem deals with *eagerness* of asynchronous tasks. Imagine that we put in the location a rewrite to another location: handler *getUrl* will run before redirection, but variable *hs_async_httpbin* will never be used because we'll get out from the current location.

The task starts from the content handler asynchronously, and the lazy bytestring — the contents — gets used in the task as is, with all of its originally computed chunks.

## Examples (including online image converter)

Let's rewrite our *timer* example using *haskell_async_content*.

**File test.hs** (*additions*)

```
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE MagicHash #-}


-- ...


import           GHC.Prim
import           Data.ByteString.Unsafe
import           Data.ByteString.Internal (accursedUnutterablePerformIO)


-- ...


packLiteral :: Int -> GHC.Prim.Addr# -> ByteString
packLiteral l s = accursedUnutterablePerformIO $ unsafePackAddressLen l s

delayContent :: ByteString -> IO ContentHandlerResult
delayContent v = do
    v' <- delay v
    return $ (, packLiteral 10 "text/plain"#, 200) $
        L.concat ["Waited ", v', " sec\n"]
ngxExportAsyncHandler 'delayContent
```

For the *content type* we used a static string *"text/plain"#* that ends with a *magic hash* merely to avoid any dynamic memory allocations.

**File test.conf** (*additions*)

```
        location /timer/ch {
            haskell_run_async_on_request_body reqFld $hs_timeout timer;
            haskell_async_content delayContent $hs_timeout;
        }
```

Run curl tests.

```
  curl -d 'timer=3' 'http://127.0.0.1:8010/timer/ch'
  Waited 3 sec
  curl 'http://127.0.0.1:8010/timer/ch'
  Waited 0 sec
```

In the next example we will create an *online image converter* to convert images of various formats into PNG using Haskell library *JuicyPixels*.

**File test.hs** (*additions*)

```
import             Codec.Picture

-- ...

convertToPng :: L.ByteString -> ByteString -> IO ContentHandlerResult
convertToPng t = const $ return $
    case decodeImage $ L.toStrict t of
        Left e -> (C8L.pack e, packLiteral 10 "text/plain"#, 500)
        Right image -> case encodeDynamicPng image of
                Left e -> (C8L.pack e, packLiteral 10 "text/plain"#, 500)
                Right png -> (png, packLiteral 9 "image/png"#, 200)
ngxExportAsyncHandlerOnReqBody 'convertToPng
```

We are going to run instances of *convertToPng* on multiple CPU cores, and therefore it's better now to compile this with option *-feager-blackholing*.

```
  ghc -O2 -feager-blackholing -dynamic -shared -fPIC -L$(ghc --print-libdir↙
    ↘ )/rts -lHSrts_thr-ghc$(ghc --numeric-version) test.hs -o test.so
  [1 of 1] Compiling NgxHaskellUserRuntime ( test.hs, test.o )
  Linking test.so ...
  cp test.so /var/lib/nginx/
```

**File test.conf** (*additions*)

```
    haskell rts_options -N4 -A32m -qg;

    limit_conn_zone all zone=all:10m;

    # ...

        location /convert/topng {
            limit_conn all 4;
```

```
        client_max_body_size 20m;
        haskell_request_body_read_temp_file on;
        haskell_async_content_on_request_body convertToPng;
    }
```

Directive *haskell rts_options* declares that we are going to use 4 CPU cores (*-N4*) for image conversion tasks: this is a good choice on a quad-core processor when high CPU utilization is expected. For dealing with huge images, we also increased Haskell GC allocation area up to *32Mb* (*-A32m*) to possibly minimize frequency of GC calls. We also forcibly switched to sequential GC (*-qg*), which is quite appropriate in our intrinsically single-threaded handler *convertToPng*. Directives *limit_conn_zone* and *limit_conn* must effectively limit number of simultaneously processed client requests to the number of CPU cores (*4*) in order to protect the CPU from overloading.

In location */convert/topng*, directive *client_max_body_size* declares that all requests whose bodies exceed *20Mb* will be rejected. Directive *haskell_request_body_read_temp_file on* makes the Haskell part able to read huge request bodies that have been buffered in a temporary file by Nginx. Notice that we do not pass any value into directive *haskell_async_content_on_request_body*, therefore its second argument is simply omitted.

For running tests, an original file, say *sample.tif*, must be prepared. We will pipe command *display* from *ImageMagick* to the output of curl for more fun.

```
curl --data-binary @sample.tif 'http://127.0.0.1:8010/convert/topng' | ↲
    ↳ display
```

# Asynchronous services

Asynchronous tasks run in a request context, whereas asynchronous services run in a worker context. They start when the module gets initialized in a worker, and stop when a worker terminates. They are useful for gathering rarely changed data shared in many requests.

There is only one type of asynchronous services exporters.

| Type | Exporter |
| --- | --- |
| ByteString -> Bool -> IO L.ByteString | ngxExportServiceIOYY (NGX_EXPORT_SERVICE_IOY_Y) |

It accepts a strict bytestring and a boolean value, and returns a lazy bytestring (chunks of data). If the boolean argument is *True* then this service has never been called before in this worker process: this can be used to initialize some global data needed by the service on the first call.

Services are declared with Nginx directive *haskell_run_service*. As far as they are not bound to requests, the directive is only available on the *http* configuration level.

```
        haskell_run_service getUrlService $hs_service_httpbin "http://httpbin.org";
```

The first argument is, as ever, the name of a Haskell handler, the second — a variable where the service result will be put, and the third argument is data passed to the handler *getUrlService* in its first parameter. Notice that the third argument cannot contain variables because variable handlers in Nginx are only available in a request context, hence this argument may only be a static string.

Asynchronous services are bound to the Nginx event loop in the same way as asynchronous tasks. When a service finishes its computation, it pokes data into buffers and writes into eventfd (or a pipe's write end). Then the event handler immediately restarts the service with the boolean argument equal to *False*. This is responsibility of the author of a service handler to avoid dry runs and make sure that it is called not so often in a row. For example, if a service polls periodically, then it must delay for this time itself like in the following example.


## An example

Let's retrieve content of a specific URL, say *httpbin.org*, in background. Data will update every 20 seconds.

**File test.hs** (*additions*)

```
import            Network.HTTP.Client
import            Control.Exception
import            System.IO.Unsafe
import            Control.Monad


-- ...


httpManager :: Manager
httpManager = unsafePerformIO $ newManager defaultManagerSettings
{-# NOINLINE httpManager #-}


getUrl :: ByteString -> IO C8L.ByteString
getUrl url = catchHttpException $ getResponse url $ flip httpLbs httpManager
    where getResponse u = fmap responseBody . (parseRequest (C8.unpack u) >>=)


catchHttpException :: IO C8L.ByteString -> IO C8L.ByteString
catchHttpException = (`catch` \e ->
        return $ C8L.pack $ "HTTP EXCEPTION: " ++ show (e :: HttpException))


getUrlService :: ByteString -> Bool -> IO L.ByteString
getUrlService url firstRun = do
    unless firstRun $ threadDelay $ 20 * 1000000
    getUrl url
ngxExportServiceIOYY 'getUrlService
```

The *httpManager* defines a global state, not to say a *variable*: this is an asynchronous HTTP client implemented in module *Network.HTTP.Client*. Pragma *NOINLINE* ensures that all functions will refer to the same client object, i.e. it will nowhere be inlined. Functions *getUrl* and *catchHttpException* are used in our service handler *getUrlService*. The handler waits 20

seconds on every run except the first, and then runs the HTTP client. All HTTP exceptions are caught by *catchHttpException*, others hit the handler on top of the custom Haskell code and get logged by Nginx.

**File test.conf** (*additions*)

```
haskell_run_service getUrlService $hs_service_httpbin "http://httpbin.org";

# ...

    location /httpbin {
        echo $hs_service_httpbin;
    }
```

Run curl tests.

```
curl 'http://127.0.0.1:8010/httpbin'
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv='content-type' value='text/html;charset=utf8'>
  <meta name='generator' value='Ronn/v0.7.3 (http://github.com/rtomayko/↵
      ↳ ronn/tree/0.7.3)'>
  <title>httpbin(1): HTTP Client Testing Service</title>


...
```

This must run really fast because it shows data that has already been retrieved by the service, requests do not trigger any network activity with *httpbin.org* by themselves!

## Termination of a service

Services are killed on a worker's exit with an asynchronous exception *WorkerProcessIsExiting*. Then the worker waits *synchronously* until all of its services' threads exit, and calls *hs_exit()*. This scenario has two important implications.

1. The Haskell service handler may catch *WorkerProcessIsExiting* on exit and make persistency actions such as writing files if they are needed.
2. *Unsafe blocking* FFI calls must be avoided in service handlers as they may hang the Nginx worker, and it won't exit. Using *interruptible* FFI fixes this problem.

## Shared services

An asynchronous service may store its result in shared memory accessible from all worker processes. This is achieved with directive *haskell_service_var_in_shm*. For example, the following declaration (in *http* clause),

```
haskell_service_var_in_shm httpbin 512k /tmp $hs_service_httpbin;
```

makes service *getUrlService*, that stores its result in variable *hs_service_httpbin*, shared. The first argument of the directive — *httpbin* — is an identifier of a shared memory segment, *512k* is its maximum size, */tmp* is a directory where *file locks* will be put (see below), and *$hs_service_httpbin* is the service variable.

Shared services are called *shared* not only because they store results in shared memory, but also because at any moment of the Nginx master lifetime there is only one worker that runs a specific service. When workers start, they race to acquire a *file lock* for a service, and if a worker wins the race, it holds the lock until it exits or dies. Other workers' services of the same type wait until the lock is freed. The locks are implemented via POSIX *advisory* file locks, and so require a directory where they will be put. The directory must be *writable* to worker processes, and */tmp* seems to be a good choice in general.

## Update variables

The active shared service puts the value of the shared variable in a shared memory, services on other workers wait and do nothing else. Requests may come to any worker (with active or inactive services), fortunately the service result is shared and they can return it as is. But what if the result must be somehow interpreted by Haskell handlers before returning it in the response? Could the handlers just peek into the shared memory and do what they want with the shared data? Unfortunately, not: the shared memory is accessible for reading and writing only from the Nginx part!

Does it mean that we have only one option to let the Haskell part update its global state unavailable in inactive workers: passing values of shared variables into the Haskell part on every request? This would be extremely inefficient. Update variables is a trick to avoid this. They evaluate to the corresponding service variable's value only when it changes in the shared memory since the last check in the current worker, and to an empty string otherwise. Every service variable has its own update variable counterpart whose name is built from the service variable's name prefixed by *_upd__*.

### An example

Let's extend our example with loading a page in background. We are still going to load *httpbin.org*, but this time let's assume that we have another task, say extracting all links from the page and showing them in the response sorted. For that we could add a Haskell handler, say *sortLinks*, and pass to it all the page content on every request. But the page may appear huge, let's extract all the links from it and put them into a global state using update variable *_upd__hs_service_httpbin*. In this case function *sortLinks* must be impure, as it must be able to read from the global state.

**File test.hs** (*additions*)

```haskell
{-# LANGUAGE OverloadedStrings #-}

-- ...

import           Data.IORef
import           Text.Regex.PCRE.ByteString
import           Text.Regex.Base.RegexLike
import qualified Data.Array as A
```

```haskell
import            Data.List
import qualified Data.ByteString as B

-- ...

gHttpbinLinks :: IORef [ByteString]
gHttpbinLinks = unsafePerformIO $ newIORef []
{-# NOINLINE gHttpbinLinks #-}

grepLinks :: ByteString -> [ByteString]
grepLinks =
    map (fst . snd) . filter ((1 ==) . fst) . concatMap A.assocs .
        filter (not . null) . concatMap (matchAllText regex) .
            C8.lines
    where regex = makeRegex $ C8.pack "a href=\"([^\"]+)\"" :: Regex

grepHttpbinLinks :: ByteString -> IO L.ByteString
grepHttpbinLinks "" = return ""
grepHttpbinLinks v = do
    writeIORef gHttpbinLinks $ grepLinks $ B.copy v
    return ""
ngxExportIOYY 'grepHttpbinLinks

sortLinks :: ByteString -> IO L.ByteString
sortLinks "httpbin" =
    L.fromChunks . sort . map (`C8.snoc` '\n') <$> readIORef gHttpbinLinks
sortLinks _ = return ""
ngxExportIOYY 'sortLinks
```

Here *gHttpbinLinks* is the global state, *grepHttpbinLinks* is a handler for update variable *_upd___hs_service_httpbin*, almost all the time it does nothing — just returns an empty string, but when the update variable becomes not empty, it updates the global state and returns an empty string again. Notice that the original bytestring is copied with *B.copy* before its parts get collected as matches and put in the global state. This is an important step because the original bytestring's lifetime does not extend beyond the current request whereas the global state may last much longer! Sometimes copying is not necessary, for example when the bytestring gets deserialized into an object in-place. Handler *sortLinks* is parameterized by data identifier: when the identifier is equal to *httpbin*, it reads the global state and returns it sorted, otherwise it returns an empty string.

**File test.conf** (*additions*)

```
    haskell_service_var_in_shm httpbin 512k /tmp $hs_service_httpbin;

    # ...

        location /httpbin/sortlinks {
            haskell_run grepHttpbinLinks $_upd_links_ $_upd__hs_service_httpbin;
            haskell_run sortLinks $hs_links "${_upd_links_}httpbin";
            echo $hs_links;
        }
```

We have to pass variable *__upd_links__* in *sortLinks* because this will trigger update in the worker by *grepHttpbinLinks*, otherwise update won't run: remember that Nginx directives are lazy? On the other hand, *__upd_links__* is always empty and won't mess up with the rest of the argument — value *httpbin*.

Run curl tests.

```
curl 'http://127.0.0.1:8010/httpbin/sortlinks'
/
/absolute-redirect/6
/anything
/basic-auth/user/passwd
/brotli
/bytes/1024


...
```

## Shm stats variables

Every service variable in shared memory has another associated variable that provides basic stats in format *timestamp | size | changes | failures | failed*, where *timestamp* is a number of seconds elapsed from the beginning of the *UNIX epoch* till the last change of the variable's value, *size* is the size of the variable in bytes, *changes* is a number of changes, and *failures* is a number of memory allocation failures since the last Nginx reload, the value of flag *failed* (*0* or *1*) denotes if the last attempt of memory allocation from the shared memory pool for a new value of the variable has failed. The name of the shm stats variable is built from the service variable's name with prefix *__shm___*.

### An example

Let's add a location to show shm stats about our *httpbin* service. This time only configuration file *test.conf* is affected.

**File test.conf** (*additions*)

```
location /httpbin/shmstats {
    echo "Httpbin service shm stats: $_shm__hs_service_httpbin";
}
```

Run curl tests.

```
curl 'http://127.0.0.1:8010/httpbin/shmstats'
Httpbin service shm stats: 1516274639 | 13011 | 1 | 0 | 0
```

From this output we can find that payload size of *httpbin.org* is *13011* bytes, the service variable was updated only once (less than 20 seconds elapsed from start of Nginx), and that there were no memory allocation failures.

## Update callbacks

There is a special type of single-shot services called update callbacks. They are declared like

```
haskell_service_var_update_callback cbHttpbin $hs_service_httpbin optional_value;
```

Here *cbHttpbin* is a Haskell handler exported by *ngxExportServiceIOYY* as always. Variable *hs_service_httpbin* must be declared in directive *haskell_service_var_in_shm*. Argument *optional_value* is a string, it can be omitted, in which case handler *cbHttpbin* gets the value of service variable *hs_service_httpbin* as its first argument.

Update callbacks do not return results. They run from a worker that holds the active service on every change of the service variable, and shall be supposedly used to integrate with other Nginx modules by signaling specific Nginx locations via an HTTP client.

### An example

Let's count all changes of service variable *hs_service_httpbin* during Nginx lifetime (originally I supposed that its content won't change after the first initialization because *httpbin.org* looks like a static page, but responses appeared to be able to vary from time to time). For this we will use counters from *nginx-custom-counters-module*.

**File test.hs** (*additions*)

```
cbHttpbin :: ByteString -> Bool -> IO L.ByteString
cbHttpbin url firstRun = do
    when firstRun $ threadDelay $ 5 * 1000000
    getUrl url
ngxExportServiceIOYY 'cbHttpbin
```

Handler *cbHttpbin* is a simple HTTP client. On the first run it waits 5 seconds before sending request because the request is supposed to be destined to self, while Nginx workers may appear to be not ready to accept it.

**File test.conf** (*additions*)

```
    haskell_service_var_update_callback cbHttpbin $hs_service_httpbin
                                    "http://127.0.0.1:8010/httpbin/count";

    # ...

        location /httpbin/count {
            counter $cnt_httpbin inc;
            return 200;
        }

        location /counters {
            echo "Httpbin service changes count: $cnt_httpbin";
        }
```

Wait at least 5 seconds after Nginx start and run curl tests.

```
curl 'http://127.0.0.1:8010/counters'
Httpbin service changes count: 1
```

Further the count will probably be steadily increasing.

```
curl 'http://127.0.0.1:8010/counters'
Httpbin service changes count: 3
```

# Service hooks

Service hooks allow for interaction with running services, both per-worker and shared. They are supposed to change global states that affect services behavior and can be thought of as service API handlers, thereto being run from dedicated Nginx locations.

| Type | Exporter |
| --- | --- |
| ByteString -> IO L.ByteString | ngxExportServiceHook (NGX_EXPORT_SERVICE_HOOK) |

Service hooks install a content handler when declared. In the following example,

```
location /httpbin/url {
    haskell_service_hook getUrlServiceHook $hs_service_httpbin $arg_v;
}
```

location */httpbin/url* derives the content handler which signals all workers via an event channel upon receiving a request. Then the event handlers in all workers run the hook (*getUrlServiceHook* in our case) *synchronously*, and finally send an asynchronous exception *ServiceHookInterrupt* to the service to which the service variable from the service hook declaration (*hs_service_httpbin*) corresponds. Being run synchronously, service hooks are expected to be fast, only writing data passed to them (the value of *arg_v* in our case) into a global state. In contrast to *update variables*, this data has a longer lifetime being freed in the Haskell part when the original bytestring gets garbage collected.

## An example

Let's make it able to change the URL for the *httpbin* service in runtime. For this we must enable *getUrlService* to read from a global state where the URL value will reside.

**File test.hs** (*additions, getUrlService reimplemented*)

```
import           Data.Maybe

-- ...

getUrlServiceLink :: IORef (Maybe ByteString)
```

```haskell
getUrlServiceLink = unsafePerformIO $ newIORef Nothing
{-# NOINLINE getUrlServiceLink #-}

getUrlServiceLinkUpdated :: IORef Bool
getUrlServiceLinkUpdated = unsafePerformIO $ newIORef True
{-# NOINLINE getUrlServiceLinkUpdated #-}

getUrlService :: ByteString -> Bool -> IO L.ByteString
getUrlService url = const $ do
    url' <- fromMaybe url <$> readIORef getUrlServiceLink
    updated <- readIORef getUrlServiceLinkUpdated
    atomicWriteIORef getUrlServiceLinkUpdated False
    unless updated $ threadDelay $ 20 * 1000000
    getUrl url'
ngxExportServiceIOYY 'getUrlService

getUrlServiceHook :: ByteString -> IO L.ByteString
getUrlServiceHook url = do
    writeIORef getUrlServiceLink $ if B.null url
                                       then Nothing
                                       else Just url
    atomicWriteIORef getUrlServiceLinkUpdated True
    return $ if B.null url
                 then "getUrlService reset URL"
                 else L.fromChunks ["getUrlService set URL ", url]
ngxExportServiceHook 'getUrlServiceHook
```

Service hook *getUrlServiceHook* writes into two global states: *getUrlServiceLink* where the URL is stored, and *getUrlServiceLinkUpdated* which will signal service *getUrlService* that the URL has been updated.

**File test.conf** (*additions*)

```
    haskell_service_hooks_zone hooks 32k;

    # ...

        location /httpbin/url {
            allow 127.0.0.1;
            deny all;
            haskell_service_hook getUrlServiceHook $hs_service_httpbin $arg_v;
        }
```

Directive *haskell_service_hooks_zone* declares a shm zone where Nginx will temporarily store data for the hook (the value of *arg_v*). This directive is not mandatory: shm zone is not really needed when service hooks pass nothing. Location */httpbin/url* is protected from unauthorized access with Nginx directives *allow* and *deny*.

Run curl tests.

First let's check that *httpbin.org* replies as expected.

```
curl 'http://127.0.0.1:8010/httpbin'
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv='content-type' value='text/html;charset=utf8'>
  <meta name='generator' value='Ronn/v0.7.3 (http://github.com/rtomayko/↙
      ↳ ronn/tree/0.7.3)'>
  <title>httpbin(1): HTTP Client Testing Service</title>


...
curl 'http://127.0.0.1:8010/httpbin/sortlinks'
/
/absolute-redirect/6
/anything
/basic-auth/user/passwd
/brotli
/bytes/1024


...
```

Then change URL to, say, *example.com*,

```
curl 'http://127.0.0.1:8010/httpbin/url?v=http://example.com'
```

and peek, by the way, into the Nginx error log.

```
2018/02/13 16:12:33 [alert] 28794#0: service hook reported "getUrlService↙
    ↳  set URL http://example.com"
2018/02/13 16:12:33 [alert] 28795#0: service hook reported "getUrlService↙
    ↳  set URL http://example.com"
2018/02/13 16:12:33 [alert] 28797#0: service hook reported "getUrlService↙
    ↳  set URL http://example.com"
2018/02/13 16:12:33 [alert] 28798#0: service hook reported "getUrlService↙
    ↳  set URL http://example.com"
2018/02/13 16:12:33 [alert] 28797#0: an exception was caught while ↙
    ↳ getting value of service variable "hs_service_httpbin": "Service ↙
    ↳ was interrupted by a service hook", using old value
```

All 4 workers were signaled, and the only *active* service (remember that *getUrlService* was made *shared*) was interrupted. Do not be deceived by *using old value*: the new URL will be read in by the service from the global state immediately after restart, and the service variable will be updated.

Let's see what we are getting now.

```
curl 'http://127.0.0.1:8010/httpbin'
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
```

```
...
curl 'http://127.0.0.1:8010/httpbin/sortlinks'
http://www.iana.org/domains/example
```

Let's reset the URL.

```
curl 'http://127.0.0.1:8010/httpbin/url'
curl 'http://127.0.0.1:8010/httpbin'
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv='content-type' value='text/html;charset=utf8'>
  <meta name='generator' value='Ronn/v0.7.3 (http://github.com/rtomayko/↵
      ↳ ronn/tree/0.7.3)'>
  <title>httpbin(1): HTTP Client Testing Service</title>

...
curl 'http://127.0.0.1:8010/httpbin/sortlinks'
/
/absolute-redirect/6
/anything
/basic-auth/user/passwd
/brotli
/bytes/1024


...
```

In the log we'll find

```
2018/02/13 16:24:12 [alert] 28795#0: service hook reported "getUrlService↵
    ↳ reset URL"
2018/02/13 16:24:12 [alert] 28794#0: service hook reported "getUrlService↵
    ↳ reset URL"
2018/02/13 16:24:12 [alert] 28797#0: service hook reported "getUrlService↵
    ↳ reset URL"
2018/02/13 16:24:12 [alert] 28798#0: service hook reported "getUrlService↵
    ↳ reset URL"
2018/02/13 16:24:12 [alert] 28797#0: an exception was caught while ↵
    ↳ getting value of service variable "hs_service_httpbin": "Service ↵
    ↳ was interrupted by a service hook", using old value
```

## Service update hooks

This is a reimplementation of *update variables* for shared services by means of service hooks.
Update hooks have a number of advantages over update variables.

1. No need for obscure treatment of update variables in configuration files.
2. No need for copying the original argument: its data is freed in the Haskell part.
3. Nginx don't need to access shared memory on every single request for checking if the
   service data has been altered.

There is a subtle difference with update variables though. As soon as with update hooks new service variable data is propagated to worker processes asynchronously via an event channel, there always exists a very short transient period between the moments when the service variable gets altered in shared memory and the global state gets updated in a worker, during which events related to client requests may occur.

An update hook is exported with exporter *ngxExportServiceHook*, and declared using directive *haskell_service_update_hook* on the *http* configuration level.

### An example

Let's reimplement the example with update of service links using a service hook.

**File test.hs** (*additions*)

```haskell
grepHttpbinLinksHook :: ByteString -> IO L.ByteString
grepHttpbinLinksHook v = do
    let links = grepLinks v
        linksList = let ls = B.intercalate " " links
                    in if B.null ls
                           then "<NULL>"
                           else ls
    writeIORef gHttpbinLinks links
    return $ L.fromChunks ["getUrlService set links ", linksList]
ngxExportServiceHook 'grepHttpbinLinksHook
```

**File test.conf** (*additions*)

```
    haskell_service_update_hook grepHttpbinLinksHook $hs_service_httpbin;

    # ...

        location /httpbin/sortlinks/hook {
            haskell_run sortLinks $hs_links httpbin;
            echo $hs_links;
        }
```

For testing this, watch the Nginx error log and change the URL of the service with requests to location */httpbin/url* like in the previous example.

## C plugins with low level access to Nginx objects

Serialized pointer to the Nginx *request object* is accessible via a special variable *_r_ptr*. Haskell handlers have no benefit from this because they do not know how the request object is built. However they may run C code having been compiled with this knowledge. The low level access to the Nginx request object makes it possible to do things that are not feasible to do without this. As soon as a C plugin can do whatever a usual Nginx module can, using it from a Haskell handler must be very cautious. All synchronous and asynchronous Haskell handlers can access the Nginx request object and pass it to a C plugin. Using it in a C plugin which runs in asynchronous context has not been investigated and is probably dangerous in many aspects,

with exception (probably) of read-only access. After all, an Nginx worker is a single-threaded process, and the standard Nginx tools and APIs were not designed for using in multi-threaded environments. As such, using C plugins in asynchronous Haskell handlers must be regarded strictly as experimental!

## An example

Let's write a plugin that will add an HTTP header to the response.

**File test\_c\_plugin.h**

```c
#ifndef NGX_HTTP_HASKELL_TEST_C_PLUGIN_H
#define NGX_HTTP_HASKELL_TEST_C_PLUGIN_H

#include <ngx_core.h>
#include <ngx_http.h>

ngx_int_t ngx_http_haskell_test_c_plugin(ngx_http_request_t *r);

#endif
```

**File test\_c\_plugin.c**

```c
#include "test_c_plugin.h"

static const ngx_str_t haskell_module = ngx_string("Nginx Haskell module");

ngx_int_t
ngx_http_haskell_test_c_plugin(ngx_http_request_t *r)
{
    ngx_table_elt_t  *x_powered_by;

    if (r == NULL) {
        return NGX_ERROR;
    }

    x_powered_by = ngx_list_push(&r->headers_out.headers);

    if (!x_powered_by) {
        ngx_log_error(NGX_LOG_CRIT, r->connection->log, 0,
                      "Unable to allocate memory to set X-Powered-By header");
        return NGX_ERROR;
    }

    x_powered_by->hash = 1;
    ngx_str_set(&x_powered_by->key, "X-Powered-By");
    x_powered_by->value = haskell_module;

    return NGX_OK;
}
```

Notice that the request object *r* gets checked in function *ngx\_http\_haskell\_test\_c\_plugin()*

against the *NULL* value. Normally in an Nginx C code this check is redundant, however in our plugin this is important because serialization of the request object may fail, and in this case the Nginx module will serialize a null pointer.

Let's compile the C code. For this we need a directory where Nginx sources were sometime compiled. Let's refer to it in an environment variable *NGX_HOME*.

```
NGX_HOME=/path/to/nginx_sources
```

Here we are going to mimic the Nginx build process.

```
gcc -O2 -fPIC -c -o test_c_plugin.o -I $NGX_HOME/src/core -I $NGX_HOME/↙
    ↳ src/http -I $NGX_HOME/src/http/modules -I $NGX_HOME/src/event -I ↙
    ↳ $NGX_HOME/src/evwnt/modules -I $NGX_HOME/src/os/unix -I $NGX_HOME/↙
    ↳ objs test_c_plugin.c
```

Now we have an object file *test_c_plugin.o* to link with the Haskell code. Below is the Haskell code itself.

**File test.hs** (*additions*)

```
import          Data.Binary.Get
import          Foreign.C.Types
import          Foreign.Ptr


-- ...


foreign import ccall unsafe "test_c_plugin.h ngx_http_haskell_test_c_plugin"
    test_c_plugin :: Ptr () -> IO CIntPtr

toRequestPtr :: ByteString -> Ptr ()
toRequestPtr = wordPtrToPtr . fromIntegral . runGet getWordhost . L.fromStrict

testCPlugin :: ByteString -> IO L.ByteString
testCPlugin v = do
    res <- test_c_plugin $ toRequestPtr v
    return $ if res == 0
                then "Success!"
                else "Failure!"
ngxExportIOYY 'testCPlugin
```

Handler *testCPlugin* runs function *ngx_http_haskell_test_c_plugin()* from the C plugin and returns *Success!* or *Failure!* in cases when the C function returns *NGX_OK* or *NGX_ERROR* respectively. When compiled with *ghc*, this code has to be linked with *test_c_plugin.o*.

```
ghc -O2 -dynamic -shared -fPIC -L$(ghc --print-libdir)/rts ↙
    ↳ -lHSrts_thr-ghc$(ghc --numeric-version) test_c_plugin.o test.hs -o ↙
    ↳ test.so
[1 of 1] Compiling NgxHaskellUserRuntime ( test.hs, test.o )
Linking test.so ...
cp test.so /var/lib/nginx/
```

**File test.conf** (*additions*)

```
        location /cplugin {
            haskell_run testCPlugin $hs_test_c_plugin $_r_ptr;
            echo "Test C plugin returned $hs_test_c_plugin";
        }
```

Run curl tests.

```
  curl -D- 'http://localhost:8010/cplugin'
  HTTP/1.1 200 OK
  Server: nginx/1.12.1
  Date: Thu, 08 Mar 2018 12:09:52 GMT
  Content-Type: application/octet-stream
  Transfer-Encoding: chunked
  Connection: keep-alive
  X-Powered-By: Nginx Haskell module

  Test C plugin returned Success!
```

The header *X-Powered-By* is in the response!

Notice that the value of *__r_ptr* has a binary representation, and therefore must not be used in textual contexts such as Haskell *data* declarations and JSON objects.

## C plugins in service update hooks

Service update hooks can be used to replace service *update callbacks*. Indeed, being run *synchronously* from an event handler, a service hook could safely call a C function which would acquire related to Nginx context from Nginx global variables such as *ngx_cycle* for doing a variety of low level actions.

Below is a table of functions exported from the Haskell module that return opaque pointers to Nginx global variables for using them in C plugins.

| Function | Returned value and its type |
|---|---|
| ngxCyclePtr | value of argument `cycle` in the worker's initialization function (of type `ngx_cycle_t *`) |
| ngxUpstreamMainConfPtr | value of expression `ngx_http_cycle_get_module_main_conf(cycle, ngx_http_upstream_module)` in the worker's initialization function (of type `ngx_http_upstream_main_conf_t *`) |
| ngxCachedTimePtr | *address* of the Nginx global variable `ngx_cached_time` (of type `volatile ngx_time_t **`) |

Notice that besides synchronous nature of service update hooks, there are other features that distinguish them from service update callbacks.

1. As soon as running C plugins can be useful not only in shared services, but in normal per-worker services too, service update hooks are allowed in both the types.
2. Unlike update callbacks, service hooks get triggered in all worker processes.
3. Unlike update callbacks, service hooks get triggered even when the value of the service variable has not been actually changed.

**An example**

See implementation of *nginx-healthcheck-plugin*.

# Efficiency of data exchange between Nginx and Haskell handlers

Haskell handlers may accept strings (`String` or `[String]`) and *strict* bytestrings (`ByteString`), and return strings, *lazy* bytestrings and booleans. Input C-strings are marshaled into a *String* with *peekCStringLen* which has linear complexity $O(n)$, output *Strings* are marshaled into C-strings with *newCStringLen* which is also $O(n)$. The new C-strings get freed upon the request termination in the Nginx part.

The bytestring counterparts are much faster. Both input and output are $O(1)$, using *unsafePackCStringLen* and a Haskell *stable pointer* to lazy bytestring buffers created inside Haskell handlers. If an output lazy bytestring has more than one chunk, a new single-chunked C-string will be created in variable and service handlers, but not in content handlers because the former use the chunks directly when constructing contents. Holding a stable pointer to a bytestring's chunks in the Nginx part ensures that they won't be garbage collected until the pointer gets freed. Stable pointers get freed upon the request termination for variable and content handlers, and before the next service iteration for service handlers.

Complex scenarios may require *typed exchange* between Haskell handlers and the Nginx part using *serialized* data types such as Haskell records. In this case *bytestring* flavors of the handlers would be the best choice. There are two well-known serialization mechanisms: *packing Show / unpacking Read* and *ToJSON / FromJSON* from Haskell package *aeson*. In practice, *Show* is basically faster than *ToJSON*, however in many cases *FromJSON* outperforms *Read*.

A variable handler of a shared service makes a copy of the variable's value because shared data can be altered by any worker at any moment, and there is no safe way to hold a reference to a shared data without locking. In contrast, a variable handler of a normal per-worker service shares a reference to the value with the service. Obviously, this is still not safe. Imagine that some request gets a reference to a service value from the variable handler, then lasts some time and later uses this reference again: the reference could probably be freed by this time because the service could have altered its data since the beginning of the request. This catastrophic scenario could have been avoided by using a copy of the service value in every request like in shared services, but this would unnecessarily hit performance, therefore requests share *counted references* to service values, and as soon as the count reaches *0*, the service value gets freed.

# Exceptions in Haskell handlers

There is no way to catch exceptions in *pure* handlers. However they can arise from using *partial* functions such as *head* and *tail*! Switching to their *total* counterparts from module *Safe* can mitigate this issue, but it is not possible to eliminate it completely.

Fortunately, all exceptions, synchronous and asynchronous, are caught on top of the module's Haskell code. If a handler does not catch an exception itself, the exception gets caught higher and logged by Nginx. However, using exception handlers in Haskell handlers, when it's possible, should be preferred.

# Summary table of all Nginx directives of the module

| Directive | Level | Comment |
| --- | --- | --- |
| `haskell compile` | `http` | Compile Haskell code found in the last argument. Accepts arguments *threaded* (use *threaded* RTS library), *debug* (use *debug* RTS library), and *standalone* (use *standalone* approach). |
| `haskell load` | `http` | Load the specified Haskell library. |
| `haskell ghc_extra_options` | `http` | Specify extra options for GHC when the library compiles. |
| `haskell rts_options` | `http` | Specify options for Haskell RTS. |
| `haskell program_options` | `http` | Specify program options. This is just another way for passing data into Haskell handlers. |
| `haskell_run` | `server,` `location,` `location if` | Run a synchronous Haskell task. |
| `haskell_run_async` | `server,` `location,` `location if` | Run an asynchronous Haskell task. |
| `haskell_run_async_on_request_body` | `server,` `location,` `location if` | Run an asynchronous Haskell request body handler. |
| `haskell_run_service` | `http` | Run a Haskell service. |
| `haskell_service_var_update_callback` | `http` | Run a callback on a service variable's update. |

| Directive | Level | Comment |
|---|---|---|
| `haskell_content` | `location,` `location if` | Declare a Haskell content handler. |
| `haskell_static_content` | `location,` `location if` | Declare a static Haskell content handler. |
| `haskell_unsafe_content` | `location,` `location if` | Declare an unsafe Haskell content handler. |
| `haskell_async_content` | `location,` `location if` | Declare an asynchronous Haskell content handler. |
| `haskell_async_content_on_request_body` | `location,` `location if` | Declare an asynchronous Haskell content handler with access to request body. |
| `haskell_service_hook` | `location,` `location if` | Declare a service hook and create a content handler for managing the corresponding service. |
| `haskell_service_update_hook` | `http` | Declare a service update hook. |
| `haskell_request_body_read_temp_file` | `server,` `location,` `location if` | This flag (*on* or *off*) makes asynchronous tasks and content handlers read buffered in a *temporary file* POST data. If not set, then buffered data is not read. |
| `haskell_var_nocacheable` | `http` | All variables in the list become no cacheable and safe for using in ad-hoc iterations over *error_page* cycles. |
| `haskell_var_compensate_uri_changes` | `http` | All variables in the list allow to cheat *error_page* when used in its redirections and make the cycle infinite. |
| `haskell_service_var_ignore_empty` | `http` | Do not write the service result when its value is empty. |
| `haskell_service_var_in_shm` | `http` | Store the service result in a shared memory. Implicitly declares a shared service. |
| `haskell_service_hooks_zone` | `http` | Declare shm zone for a temporary storage of service hooks data. |
| `haskell_request_variable_name` | `http` | Change the name of the request variable if default value *__r_ptr* is already used. |

# Module NgxExport.Tools

Package *ngx-export-tools* provides module *NgxExport.Tools* that exports various utility functions and data as well as specialized service exporters and adapters. As soon as the module is well documented, its features are only basically lined up below.

- Utility functions *terminateWorkerProcess* and *restartWorkerProcess* make it possible to terminate the worker process from within a Haskell service. Function *finalizeHTTPRequest* finalizes the current HTTP request from an asynchronous Haskell handler with the specified HTTP status and body. Function *ngxRequestPtr* unmarshals the value of Nginx variable *_r_ptr*. Function *ngxNow* returns the current time cached inside the Nginx core.
- Data *TimeInterval* and utility functions *toSec* and *threadDelaySec* can be used to specify time delays for services.
- A number of converters from custom types deriving or implementing instances of *Read* and *FromJSON* (*readFromBytestring* and friends).
- Special service exporters (*simple services*) combine various *sleeping* strategies and typing policies of services and can be used to avoid usual boilerplate code needed in the vanilla service exporters from module *NgxExport*.
- Special service adapters (*split services*) allow for distinguishing between *ignition* services (those that run when the service runs for the first time) and *deferred* services (those that run when the service runs for the second time and later).

# Appendix

## File *test.hs*

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE MagicHash #-}
{-# LANGUAGE OverloadedStrings #-}

module NgxHaskellUserRuntime where

import           NgxExport
import qualified Data.Char as C
import           Data.ByteString (ByteString)
import qualified Data.ByteString.Lazy as L
import qualified Data.ByteString.Char8 as C8
import qualified Data.ByteString.Lazy.Char8 as C8L
import           Control.Concurrent
import           Safe
import           GHC.Prim
import           Data.ByteString.Unsafe
import           Data.ByteString.Internal (accursedUnutterablePerformIO)
import           Codec.Picture
import           Network.HTTP.Client
import           Control.Exception
import           System.IO.Unsafe
import           Control.Monad
import           Data.IORef
import           Text.Regex.PCRE.ByteString
import           Text.Regex.Base.RegexLike
import qualified Data.Array as A
import           Data.List
import qualified Data.ByteString as B
import           Data.Maybe
import           Data.Binary.Get
import           Foreign.C.Types
import           Foreign.Ptr

toUpper :: String -> String
toUpper = map C.toUpper
ngxExportSS 'toUpper


ngxExportSS 'reverse


isInList :: [String] -> Bool
isInList [] = False
isInList (x : xs) = x `elem` xs
ngxExportBLS 'isInList


echo :: ByteString -> L.ByteString
echo = L.fromStrict
```

```haskell
ngxExportDefHandler 'echo

reqFld :: L.ByteString -> ByteString -> IO L.ByteString
reqFld a fld = return $ maybe C8L.empty C8L.tail $
    lookup (C8L.fromStrict fld) $ map (C8L.break (== '=')) $ C8L.split '&' a
ngxExportAsyncOnReqBody 'reqFld

delay :: ByteString -> IO L.ByteString
delay v = do
    let t = readDef 0 $ C8.unpack v
    threadDelay $ t * 1000000
    return $ C8L.pack $ show t
ngxExportAsyncIOYY 'delay

packLiteral :: Int -> GHC.Prim.Addr# -> ByteString
packLiteral l s = accursedUnutterablePerformIO $ unsafePackAddressLen l s

delayContent :: ByteString -> IO ContentHandlerResult
delayContent v = do
    v' <- delay v
    return $ (, packLiteral 10 "text/plain"#, 200) $
        L.concat ["Waited ", v', " sec\n"]
ngxExportAsyncHandler 'delayContent

convertToPng :: L.ByteString -> ByteString -> IO ContentHandlerResult
convertToPng t = const $ return $
    case decodeImage $ L.toStrict t of
        Left e -> (C8L.pack e, packLiteral 10 "text/plain"#, 500)
        Right image -> case encodeDynamicPng image of
                Left e -> (C8L.pack e, packLiteral 10 "text/plain"#, 500)
                Right png -> (png, packLiteral 9 "image/png"#, 200)
ngxExportAsyncHandlerOnReqBody 'convertToPng

httpManager :: Manager
httpManager = unsafePerformIO $ newManager defaultManagerSettings
{-# NOINLINE httpManager #-}

getUrl :: ByteString -> IO C8L.ByteString
getUrl url = catchHttpException $ getResponse url $ flip httpLbs httpManager
    where getResponse u = fmap responseBody . (parseRequest (C8.unpack u) >>=)

catchHttpException :: IO C8L.ByteString -> IO C8L.ByteString
catchHttpException = ('catch' \e ->
        return $ C8L.pack $ "HTTP EXCEPTION: " ++ show (e :: HttpException))

getUrlServiceLink :: IORef (Maybe ByteString)
getUrlServiceLink = unsafePerformIO $ newIORef Nothing
{-# NOINLINE getUrlServiceLink #-}

getUrlServiceLinkUpdated :: IORef Bool
getUrlServiceLinkUpdated = unsafePerformIO $ newIORef True
{-# NOINLINE getUrlServiceLinkUpdated #-}
```

```haskell
getUrlService :: ByteString -> Bool -> IO L.ByteString
getUrlService url = const $ do
    url' <- fromMaybe url <$> readIORef getUrlServiceLink
    updated <- readIORef getUrlServiceLinkUpdated
    atomicWriteIORef getUrlServiceLinkUpdated False
    unless updated $ threadDelay $ 20 * 1000000
    getUrl url'
ngxExportServiceIOYY 'getUrlService

getUrlServiceHook :: ByteString -> IO L.ByteString
getUrlServiceHook url = do
    writeIORef getUrlServiceLink $ if B.null url
                                       then Nothing
                                       else Just url
    atomicWriteIORef getUrlServiceLinkUpdated True
    return $ if B.null url
                 then "getUrlService reset URL"
                 else L.fromChunks ["getUrlService set URL ", url]
ngxExportServiceHook 'getUrlServiceHook

gHttpbinLinks :: IORef [ByteString]
gHttpbinLinks = unsafePerformIO $ newIORef []
{-# NOINLINE gHttpbinLinks #-}

grepLinks :: ByteString -> [ByteString]
grepLinks =
    map (fst . snd) . filter ((1 ==) . fst) . concatMap A.assocs .
        filter (not . null) . concatMap (matchAllText regex) .
            C8.lines
    where regex = makeRegex $ C8.pack "a href=\"([^\"]+)\"" :: Regex

grepHttpbinLinks :: ByteString -> IO L.ByteString
grepHttpbinLinks "" = return ""
grepHttpbinLinks v = do
    writeIORef gHttpbinLinks $ grepLinks $ B.copy v
    return ""
ngxExportIOYY 'grepHttpbinLinks

sortLinks :: ByteString -> IO L.ByteString
sortLinks "httpbin" =
    L.fromChunks . sort . map (`C8.snoc` '\n') <$> readIORef gHttpbinLinks
sortLinks _ = return ""
ngxExportIOYY 'sortLinks

cbHttpbin :: ByteString -> Bool -> IO L.ByteString
cbHttpbin url firstRun = do
    when firstRun $ threadDelay $ 5 * 1000000
    getUrl url
ngxExportServiceIOYY 'cbHttpbin

grepHttpbinLinksHook :: ByteString -> IO L.ByteString
grepHttpbinLinksHook v = do
```

```haskell
        let links = grepLinks v
            linksList = let ls = B.intercalate " " links
                        in if B.null ls
                             then "<NULL>"
                             else ls
    writeIORef gHttpbinLinks links
    return $ L.fromChunks ["getUrlService set links ", linksList]
ngxExportServiceHook 'grepHttpbinLinksHook

foreign import ccall unsafe "test_c_plugin.h ngx_http_haskell_test_c_plugin"
    test_c_plugin :: Ptr () -> IO CIntPtr

toRequestPtr :: ByteString -> Ptr ()
toRequestPtr = wordPtrToPtr . fromIntegral . runGet getWordhost . L.fromStrict

testCPlugin :: ByteString -> IO L.ByteString
testCPlugin v = do
    res <- test_c_plugin $ toRequestPtr v
    return $ if res == 0
                then "Success!"
                else "Failure!"
ngxExportIOYY 'testCPlugin
```

## File *test.conf*

```
user                 nginx;
worker_processes     4;

events {
    worker_connections  1024;
}

http {
    default_type       application/octet-stream;
    sendfile           on;

    haskell load /var/lib/nginx/test.so;

    # Use 4 cores (-N4) and a large GC allocation area (-A32m), and force
    # sequential GC (-qg) for image conversion tasks.
    #haskell rts_options -N4 -A32m -qg;

    limit_conn_zone all zone=all:10m;

    haskell_run_service getUrlService $hs_service_httpbin "http://httpbin.org";

    haskell_service_var_in_shm httpbin 512k /tmp $hs_service_httpbin;

    haskell_service_var_update_callback cbHttpbin $hs_service_httpbin
                                "http://127.0.0.1:8010/httpbin/count";
```

```
    haskell_service_hooks_zone hooks 32k;

    haskell_service_update_hook grepHttpbinLinksHook $hs_service_httpbin;

    server {
        listen          8010;
        server_name     main;

        location / {
            haskell_run toUpper $hs_upper $arg_u;
            haskell_run reverse $hs_reverse $arg_r;
            haskell_run isInList $hs_isInList $arg_a $arg_b $arg_c $arg_d;
            echo "toUpper $arg_u = $hs_upper";
            echo "reverse $arg_r = $hs_reverse";
            echo "$arg_a `isInList` [$arg_b, $arg_c, $arg_d] = $hs_isInList";
        }

        location /ch {
            haskell_run toUpper $hs_upper $arg_u;
            haskell_run reverse $hs_reverse $arg_r;
            haskell_run isInList $hs_isInList $arg_a $arg_b $arg_c $arg_d;
            haskell_content echo
"toUpper $arg_u = $hs_upper
reverse $arg_r = $hs_reverse
$arg_a `isInList` [$arg_b, $arg_c, $arg_d] = $hs_isInList
";
        }

        location /timer {
            haskell_run_async_on_request_body reqFld $hs_timeout timer;
            haskell_run_async delay $hs_waited $hs_timeout;
            echo "Waited $hs_waited sec";
        }

        location /timer/ch {
            haskell_run_async_on_request_body reqFld $hs_timeout timer;
            haskell_async_content delayContent $hs_timeout;
        }

        location /convert/topng {
            limit_conn all 4;
            client_max_body_size 20m;
            haskell_request_body_read_temp_file on;
            haskell_async_content_on_request_body convertToPng;
        }

        location /httpbin {
            echo $hs_service_httpbin;
        }

        location /httpbin/sortlinks {
            haskell_run grepHttpbinLinks $_upd_links_ $_upd__hs_service_httpbin;
```

```
            haskell_run sortLinks $hs_links "${_upd_links_}httpbin";
            echo $hs_links;
        }

        location /httpbin/sortlinks/hook {
            haskell_run sortLinks $hs_links httpbin;
            echo $hs_links;
        }

        location /httpbin/shmstats {
            echo "Httpbin service shm stats: $_shm__hs_service_httpbin";
        }

        location /httpbin/url {
            allow 127.0.0.1;
            deny all;
            haskell_service_hook getUrlServiceHook $hs_service_httpbin $arg_v;
        }

        # Counters require Nginx module nginx-custom-counters-module,
        # enable the next 2 locations if your Nginx build has support for them.

        #location /httpbin/count {
            #counter $cnt_httpbin inc;
            #return 200;
        #}

        #location /counters {
            #echo "Httpbin service changes count: $cnt_httpbin";
        #}

        location /cplugin {
            haskell_run testCPlugin $hs_test_c_plugin $_r_ptr;
            echo "Test C plugin returned $hs_test_c_plugin";
        }
    }
}
```

## File *test_c_plugin.h*

```
/* Compile:
 *      NGX_HOME=/path/to/nginx_sources
 *      gcc -fPIC -c -o test_c_plugin.o \
 *          -I $NGX_HOME/src/core \
 *          -I $NGX_HOME/src/http \
 *          -I $NGX_HOME/src/http/modules \
 *          -I $NGX_HOME/src/event \
 *          -I $NGX_HOME/src/evwnt/modules \
 *          -I $NGX_HOME/src/os/unix \
 *          -I $NGX_HOME/objs test_c_plugin.c
```

```
  */

#ifndef NGX_HTTP_HASKELL_TEST_C_PLUGIN_H
#define NGX_HTTP_HASKELL_TEST_C_PLUGIN_H

#include <ngx_core.h>
#include <ngx_http.h>

ngx_int_t ngx_http_haskell_test_c_plugin(ngx_http_request_t *r);

#endif
```

## File *test_c_plugin.c*

```
#include "test_c_plugin.h"

static const ngx_str_t haskell_module = ngx_string("Nginx Haskell module");

ngx_int_t
ngx_http_haskell_test_c_plugin(ngx_http_request_t *r)
{
    ngx_table_elt_t  *x_powered_by;

    if (r == NULL) {
        return NGX_ERROR;
    }

    x_powered_by = ngx_list_push(&r->headers_out.headers);

    if (!x_powered_by) {
        ngx_log_error(NGX_LOG_CRIT, r->connection->log, 0,
                      "Unable to allocate memory to set X-Powered-By header");
        return NGX_ERROR;
    }

    x_powered_by->hash = 1;
    ngx_str_set(&x_powered_by->key, "X-Powered-By");
    x_powered_by->value = haskell_module;

    return NGX_OK;
}
```