**Emiller's Guide To Nginx Module Development**

By [Evan Miller](http://www.evanmiller.org/)

First published: April 28, 2007 (Last edit: January 16, 2013 – [changes](http://www.evanmiller.org/nginx-modules-guide.html#changes))

Bruce Wayne: *What's that?*  
Lucius Fox: *The Tumbler? Oh… you wouldn't be interested in that.*

To fully appreciate Nginx, the web server, it helps to understand Batman, the comic book character.

Batman is fast. Nginx is fast. Batman fights crime. Nginx fights wasted CPU cycles and memory leaks. Batman performs well under pressure. Nginx, for its part, excels under heavy server loads.

But Batman would be almost nothing without the **Batman utility belt**.



**Figure 1**: The Batman utility belt, gripping Christian Bale's love handles.

At any given time, Batman's utility belt might contain a lock pick, several batarangs, bat-cuffs, a bat-tracer, bat-darts, night vision goggles, thermite grenades, smoke pellets, a flashlight, a kryptonite ring, an acetylene torch, or an Apple iPhone. When Batman needs to tranquilize, blind, deafen, stun, track, stop, smoke out, or text-message the enemy, you better believe he's reaching down for his bat-belt. The belt is so crucial to Batman's operations that if Batman had to choose between wearing pants and wearing the utility belt, he would definitely choose the belt. In fact, he \*did\* choose the utility belt, and that's why Batman wears rubber tights instead of pants (Fig. 1).

Instead of a utility belt, Nginx has a **module chain**. When Nginx needs to gzip or chunk-encode a response, it whips out a module to do the work. When Nginx blocks access to a resource based on IP address or HTTP auth credentials, a module does the deflecting. When Nginx communicates with Memcache or FastCGI servers, a module is the walkie-talkie.

Batman's utility belt holds a lot of doo-hickeys, but occasionally Batman needs a new tool. Maybe there's a new enemy against whom bat-cuffs and batarangs are ineffectual. Or Batman needs a new ability, like being able to breathe underwater. That's when Batman rings up **Lucius Fox** to engineer the appropriate bat-gadget.



**Figure 2**: Bruce Wayne (née Batman) consults with his engineer, Lucius Fox

The purpose of this guide is to teach you the details of Nginx's module chain, so that you may be like Lucius Fox. When you're done with the guide, you'll be able to design and produce high-quality modules that enable Nginx to do things it couldn't do before. Nginx's module system has a lot of nuance and nitty-gritty, so you'll probably want to refer back to this document often. I have tried to make the concepts as clear as possible, but I'll be blunt, writing Nginx modules can still be hard work.

But whoever said making bat-tools would be easy?

**Table of Contents**

1. [Prerequisites](http://www.evanmiller.org/nginx-modules-guide.html#prerequisites)
2. [High-Level Overview of Nginx's Module Delegation](http://www.evanmiller.org/nginx-modules-guide.html#overview)
3. [Components of an Nginx Module](http://www.evanmiller.org/nginx-modules-guide.html#components)
   1. [Module Configuration Struct(s)](http://www.evanmiller.org/nginx-modules-guide.html#configuration-structs)
   2. [Module Directives](http://www.evanmiller.org/nginx-modules-guide.html#directives)
   3. [The Module Context](http://www.evanmiller.org/nginx-modules-guide.html#context)
      1. [create\_loc\_conf](http://www.evanmiller.org/nginx-modules-guide.html#create_loc_conf)
      2. [merge\_loc\_conf](http://www.evanmiller.org/nginx-modules-guide.html#merge_loc_conf)
   4. [The Module Definition](http://www.evanmiller.org/nginx-modules-guide.html#definition)
   5. [Module Installation](http://www.evanmiller.org/nginx-modules-guide.html#installation)
4. [Handlers](http://www.evanmiller.org/nginx-modules-guide.html#handlers)
   1. [Anatomy of a Handler (Non-proxying)](http://www.evanmiller.org/nginx-modules-guide.html#non-proxying)
      1. [Getting the location configuration](http://www.evanmiller.org/nginx-modules-guide.html#non-proxying-config)
      2. [Generating a response](http://www.evanmiller.org/nginx-modules-guide.html#non-proxying-response)
      3. [Sending the header](http://www.evanmiller.org/nginx-modules-guide.html#non-proxying-header)
      4. [Sending the body](http://www.evanmiller.org/nginx-modules-guide.html#non-proxying-body)
   2. [Anatomy of an Upstream (a.k.a. Proxy) Handler](http://www.evanmiller.org/nginx-modules-guide.html#proxying)
      1. [Summary of upstream callbacks](http://www.evanmiller.org/nginx-modules-guide.html#proxying-summary)
      2. [The create\_request callback](http://www.evanmiller.org/nginx-modules-guide.html#create_request)
      3. [The process\_header callback](http://www.evanmiller.org/nginx-modules-guide.html#process_header)
      4. [Keeping state](http://www.evanmiller.org/nginx-modules-guide.html#keeping-state)
   3. [Handler Installation](http://www.evanmiller.org/nginx-modules-guide.html#handler-installation)
5. [Filters](http://www.evanmiller.org/nginx-modules-guide.html#filters)
   1. [Anatomy of a Header Filter](http://www.evanmiller.org/nginx-modules-guide.html#filters-header)
   2. [Anatomy of a Body Filter](http://www.evanmiller.org/nginx-modules-guide.html#filters-body)
   3. [Filter Installation](http://www.evanmiller.org/nginx-modules-guide.html#filters-installation)
6. [Load-Balancers](http://www.evanmiller.org/nginx-modules-guide.html#load_balancers)
   1. [The enabling directive](http://www.evanmiller.org/nginx-modules-guide.html#lb-directive)
   2. [The registration function](http://www.evanmiller.org/nginx-modules-guide.html#lb-registration)
   3. [The upstream initialization function](http://www.evanmiller.org/nginx-modules-guide.html#lb-upstream)
   4. [The peer initialization function](http://www.evanmiller.org/nginx-modules-guide.html#lb-peer)
   5. [The load-balancing function](http://www.evanmiller.org/nginx-modules-guide.html#lb-function)
   6. [The peer release function](http://www.evanmiller.org/nginx-modules-guide.html#lb-release)
7. [Writing and Compiling a New Nginx Module](http://www.evanmiller.org/nginx-modules-guide.html#compiling)
8. [Advanced Topics](http://www.evanmiller.org/nginx-modules-guide.html#advanced)
9. [Code References](http://www.evanmiller.org/nginx-modules-guide.html#code)

**0. Prerequisites**

You should be comfortable with C. Not just "C-syntax"; you should know your way around a struct and not be scared off by pointers and function references, and be cognizant of the preprocessor. If you need to brush up, nothing beats [K&R](http://en.wikipedia.org/wiki/The_C_Programming_Language_(book)).

Basic understanding of HTTP is useful. You'll be working on a web server, after all.

You should also be familiar with Nginx's configuration file. If you're not, here's the gist of it: there are four *contexts* (called *main*, *server*, *upstream*, and *location*) which can contain directives with one or more arguments. Directives in the main context apply to everything; directives in the server context apply to a particular host/port; directives in the upstream context refer to a set of backend servers; and directives in a location context apply only to matching web locations (e.g., "/", "/images", etc.) A location context inherits from the surrounding server context, and a server context inherits from the main context. The upstream context neither inherits nor imparts its properties; it has its own special directives that don't really apply elsewhere. I'll refer to these four contexts quite a bit, so… don't forget them.

Let's get started.

**1. High-Level Overview of Nginx's Module Delegation**

Nginx modules have three roles we'll cover:

* *handlers* process a request and produce output
* *filters* manipulate the output produced by a handler
* *load-balancers* choose a backend server to send a request to, when more than one backend server is eligible

Modules do all of the "real work" that you might associate with a web server: whenever Nginx serves a file or proxies a request to another server, there's a handler module doing the work; when Nginx gzips the output or executes a server-side include, it's using filter modules. The "core" of Nginx simply takes care of all the network and application protocols and sets up the sequence of modules that are eligible to process a request. The de-centralized architecture makes it possible for \*you\* to make a nice self-contained unit that does something you want.

Note: Unlike modules in Apache, Nginx modules are *not* dynamically linked. (In other words, they're compiled right into the Nginx binary.)

How does a module get invoked? Typically, at server startup, each handler gets a chance to attach itself to particular locations defined in the configuration; if more than one handler attaches to a particular location, only one will "win" (but a good config writer won't let a conflict happen). Handlers can return in three ways: all is good, there was an error, or it can decline to process the request and defer to the default handler (typically something that serves static files).

If the handler happens to be a reverse proxy to some set of backend servers, there is room for another type of module: the load-balancer. A load-balancer takes a request and a set of backend servers and decides which server will get the request. Nginx ships with two load-balancing modules: round-robin, which deals out requests like cards at the start of a poker game, and the "IP hash" method, which ensures that a particular client will hit the same backend server across multiple requests.

If the handler does not produce an error, the filters are called. Multiple filters can hook into each location, so that (for example) a response can be compressed and then chunked. The order of their execution is determined at compile-time. Filters have the classic "CHAIN OF RESPONSIBILITY" design pattern: one filter is called, does its work, and then calls the next filter, until the final filter is called, and Nginx finishes up the response.

The really cool part about the filter chain is that each filter doesn't wait for the previous filter to finish; it can process the previous filter's output as it's being produced, sort of like the Unix pipeline. Filters operate on *buffers*, which are usually the size of a page (4K), although you can change this in your nginx.conf. This means, for example, a module can start compressing the response from a backend server and stream it to the client before the module has received the entire response from the backend. Nice!

So to wrap up the conceptual overview, the typical processing cycle goes:

Client sends HTTP request → Nginx chooses the appropriate handler based on the location config → (if applicable) load-balancer picks a backend server → Handler does its thing and passes each output buffer to the first filter → First filter passes the output to the second filter → second to third → third to fourth → etc. → Final response sent to client

I say "typically" because Nginx's module invocation is *extremely* customizable. It places a big burden on module writers to define exactly how and when the module should run (I happen to think too big a burden). Invocation is actually performed through a series of callbacks, and there are a lot of them. Namely, you can provide a function to be executed:

* Just before the server reads the config file
* For every configuration directive for the location and server for which it appears;
* When Nginx initializes the main configuration
* When Nginx initializes the server (i.e., host/port) configuration
* When Nginx merges the server configuration with the main configuration
* When Nginx initializes the location configuration
* When Nginx merges the location configuration with its parent server configuration
* When Nginx's master process starts
* When a new worker process starts
* When a worker process exits
* When the master exits
* Handling a request
* Filtering response headers
* Filtering the response body
* Picking a backend server
* Initiating a request to a backend server
* *Re*-initiating a request to a backend server
* Processing the response from a backend server
* Finishing an interaction with a backend server

Holy mackerel! It's a bit overwhelming. You've got a lot of power at your disposal, but you can still do something useful using only a couple of these hooks and a couple of corresponding functions. Time to dive into some modules.

**2. Components of an Nginx Module**

As I said, you have a *lot* of flexibility when it comes to making an Nginx module. This section will describe the parts that are almost always present. It's intended as a guide for understanding a module, and a reference for when you think you're ready to start writing a module.

**2.1. Module Configuration Struct(s)**

Modules can define up to three configuration structs, one for the main, server, and location contexts. Most modules just need a location configuration. The naming convention for these is ngx\_http\_<module name>\_(main|srv|loc)\_conf\_t. Here's an example, taken from the dav module:

typedef struct {

ngx\_uint\_t methods;

ngx\_flag\_t create\_full\_put\_path;

ngx\_uint\_t access;

} ngx\_http\_dav\_loc\_conf\_t;

Notice that Nginx has special data types (ngx\_uint\_t and ngx\_flag\_t); these are just aliases for the primitive data types you know and love (cf. [core/ngx\_config.h](http://lxr.evanmiller.org/http/source/core/ngx_config.h#L79) if you're curious).

The elements in the configuration structs are populated by module directives.

**2.2. Module Directives**

A module's directives appear in a static array of ngx\_command\_ts. Here's an example of how they're declared, taken from a small module I wrote:

static ngx\_command\_t ngx\_http\_circle\_gif\_commands[] = {

{ ngx\_string("circle\_gif"),

NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_NOARGS,

ngx\_http\_circle\_gif,

NGX\_HTTP\_LOC\_CONF\_OFFSET,

0,

NULL },

{ ngx\_string("circle\_gif\_min\_radius"),

NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,

ngx\_conf\_set\_num\_slot,

NGX\_HTTP\_LOC\_CONF\_OFFSET,

offsetof(ngx\_http\_circle\_gif\_loc\_conf\_t, min\_radius),

NULL },

...

ngx\_null\_command

};

And here is the declaration of ngx\_command\_t (the struct we're declaring), found in [core/ngx\_conf\_file.h](http://lxr.evanmiller.org/http/source/core/ngx_conf_file.h#L77):

struct ngx\_command\_t {

ngx\_str\_t name;

ngx\_uint\_t type;

char \*(\*set)(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf);

ngx\_uint\_t conf;

ngx\_uint\_t offset;

void \*post;

};

It seems like a bit much, but each element has a purpose.

The name is the directive string, no spaces. The data type is an ngx\_str\_t, which is usually instantiated with just (e.g.) ngx\_str("proxy\_pass"). Note: an ngx\_str\_t is a struct with a data element, which is a string, and a lenelement, which is the length of that string. Nginx uses this data structure most places you'd expect a string.

type is a set of flags that indicate where the directive is legal and how many arguments the directive takes. Applicable flags, which are bitwise-OR'd, are:

* NGX\_HTTP\_MAIN\_CONF: directive is valid in the main config
* NGX\_HTTP\_SRV\_CONF: directive is valid in the server (host) config
* NGX\_HTTP\_LOC\_CONF: directive is valid in a location config
* NGX\_HTTP\_UPS\_CONF: directive is valid in an upstream config
* NGX\_CONF\_NOARGS: directive can take 0 arguments
* NGX\_CONF\_TAKE1: directive can take exactly 1 argument
* NGX\_CONF\_TAKE2: directive can take exactly 2 arguments
* …
* NGX\_CONF\_TAKE7: directive can take exactly 7 arguments
* NGX\_CONF\_FLAG: directive takes a boolean ("on" or "off")
* NGX\_CONF\_1MORE: directive must be passed at least one argument
* NGX\_CONF\_2MORE: directive must be passed at least two arguments

There are a few other options, too, see [core/ngx\_conf\_file.h](http://lxr.evanmiller.org/http/source/core/ngx_conf_file.h#L1).

The set struct element is a pointer to a function for setting up part of the module's configuration; typically this function will translate the arguments passed to this directive and save an appropriate value in its configuration struct. This setup function will take three arguments:

1. a pointer to an ngx\_conf\_t struct, which contains the arguments passed to the directive
2. a pointer to the current ngx\_command\_t struct
3. a pointer to the module's custom configuration struct

This setup function will be called when the directive is encountered. Nginx provides a number of functions for setting particular types of values in the custom configuration struct. These functions include:

* ngx\_conf\_set\_flag\_slot: translates "on" or "off" to 1 or 0
* ngx\_conf\_set\_str\_slot: saves a string as an ngx\_str\_t
* ngx\_conf\_set\_num\_slot: parses a number and saves it to an int
* ngx\_conf\_set\_size\_slot: parses a data size ("8k", "1m", etc.) and saves it to a size\_t

There are several others, and they're quite handy (see [core/ngx\_conf\_file.h](http://lxr.evanmiller.org/http/source/core/ngx_conf_file.h#L329)). Modules can also put a reference to their own function here, if the built-ins aren't quite good enough.

How do these built-in functions know where to save the data? That's where the next two elements of ngx\_command\_t come in, conf and offset. conf tells Nginx whether this value will get saved to the module's main configuration, server configuration, or location configuration (with NGX\_HTTP\_MAIN\_CONF\_OFFSET, NGX\_HTTP\_SRV\_CONF\_OFFSET, or NGX\_HTTP\_LOC\_CONF\_OFFSET). offset then specifies which part of this configuration struct to write to.

*Finally*, post is just a pointer to other crap the module might need while it's reading the configuration. It's often NULL.

The commands array is terminated with ngx\_null\_command as the last element.

**2.3. The Module Context**

This is a static ngx\_http\_module\_t struct, which just has a bunch of function references for creating the three configurations and merging them together. Its name is ngx\_http\_<module name>\_module\_ctx. In order, the function references are:

* preconfiguration
* postconfiguration
* creating the main conf (i.e., do a malloc and set defaults)
* initializing the main conf (i.e., override the defaults with what's in nginx.conf)
* creating the server conf
* merging it with the main conf
* creating the location conf
* merging it with the server conf

These take different arguments depending on what they're doing. Here's the struct definition, taken from [http/ngx\_http\_config.h](http://lxr.evanmiller.org/http/source/http/ngx_http_config.h#L22), so you can see the different function signatures of the callbacks:

typedef struct {

ngx\_int\_t (\*preconfiguration)(ngx\_conf\_t \*cf);

ngx\_int\_t (\*postconfiguration)(ngx\_conf\_t \*cf);

void \*(\*create\_main\_conf)(ngx\_conf\_t \*cf);

char \*(\*init\_main\_conf)(ngx\_conf\_t \*cf, void \*conf);

void \*(\*create\_srv\_conf)(ngx\_conf\_t \*cf);

char \*(\*merge\_srv\_conf)(ngx\_conf\_t \*cf, void \*prev, void \*conf);

void \*(\*create\_loc\_conf)(ngx\_conf\_t \*cf);

char \*(\*merge\_loc\_conf)(ngx\_conf\_t \*cf, void \*prev, void \*conf);

} ngx\_http\_module\_t;

You can set functions you don't need to NULL, and Nginx will figure it out.

Most handlers just use the last two: a function to allocate memory for location-specific configuration (called ngx\_http\_<module name>\_create\_loc\_conf), and a function to set defaults and merge this configuration with any inherited configuration (called ngx\_http\_<module name >\_merge\_loc\_conf). The merge function is also responsible for producing an error if the configuration is invalid; these errors halt server startup.

Here's an example module context struct:

static ngx\_http\_module\_t ngx\_http\_circle\_gif\_module\_ctx = {

NULL, /\* preconfiguration \*/

NULL, /\* postconfiguration \*/

NULL, /\* create main configuration \*/

NULL, /\* init main configuration \*/

NULL, /\* create server configuration \*/

NULL, /\* merge server configuration \*/

ngx\_http\_circle\_gif\_create\_loc\_conf, /\* create location configuration \*/

ngx\_http\_circle\_gif\_merge\_loc\_conf /\* merge location configuration \*/

};

Time to dig in deep a little bit. These configuration callbacks look quite similar across all modules and use the same parts of the Nginx API, so they're worth knowing about.

**2.3.1. create\_loc\_conf**

Here's what a bare-bones create\_loc\_conf function looks like, taken from the circle\_gif module I wrote (see the [the source](http://www.evanmiller.org/nginx/ngx_http_circle_gif_module.c.txt)). It takes a directive struct (ngx\_conf\_t) and returns a newly created module configuration struct (in this case ngx\_http\_circle\_gif\_loc\_conf\_t).

static void \*

ngx\_http\_circle\_gif\_create\_loc\_conf(ngx\_conf\_t \*cf)

{

ngx\_http\_circle\_gif\_loc\_conf\_t \*conf;

conf = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_circle\_gif\_loc\_conf\_t));

if (conf == NULL) {

return NGX\_CONF\_ERROR;

}

conf->min\_radius = NGX\_CONF\_UNSET\_UINT;

conf->max\_radius = NGX\_CONF\_UNSET\_UINT;

return conf;

}

First thing to notice is Nginx's memory allocation; it takes care of the free'ing as long as the module uses ngx\_palloc (a malloc wrapper) or ngx\_pcalloc (a calloc wrapper).

The possible UNSET constants are NGX\_CONF\_UNSET\_UINT, NGX\_CONF\_UNSET\_PTR, NGX\_CONF\_UNSET\_SIZE, NGX\_CONF\_UNSET\_MSEC, and the catch-all NGX\_CONF\_UNSET. UNSET tell the merging function that the value should be overridden.

**2.3.2. merge\_loc\_conf**

Here's the merging function used in the circle\_gif module:

static char \*

ngx\_http\_circle\_gif\_merge\_loc\_conf(ngx\_conf\_t \*cf, void \*parent, void \*child)

{

ngx\_http\_circle\_gif\_loc\_conf\_t \*prev = parent;

ngx\_http\_circle\_gif\_loc\_conf\_t \*conf = child;

ngx\_conf\_merge\_uint\_value(conf->min\_radius, prev->min\_radius, 10);

ngx\_conf\_merge\_uint\_value(conf->max\_radius, prev->max\_radius, 20);

if (conf->min\_radius < 1) {

ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,

"min\_radius must be equal or more than 1");

return NGX\_CONF\_ERROR;

}

if (conf->max\_radius < conf->min\_radius) {

ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,

"max\_radius must be equal or more than min\_radius");

return NGX\_CONF\_ERROR;

}

return NGX\_CONF\_OK;

}

Notice first that Nginx provides nice merging functions for different data types (ngx\_conf\_merge\_<data type>\_value); the arguments are

1. *this* location's value
2. the value to inherit if #1 is not set
3. the default if neither #1 nor #2 is set

The result is then stored in the first argument. Available merge functions include ngx\_conf\_merge\_size\_value, ngx\_conf\_merge\_msec\_value, and others. See [core/ngx\_conf\_file.h](http://lxr.evanmiller.org/http/source/core/ngx_conf_file.h#L254) for a full list.

Trivia question: How do these functions write to the first argument, since the first argument is passed in by value?

Answer: these functions are defined by the preprocessor (so they expand to a few "if" statements and assignments before reaching the compiler).

Notice also how errors are produced; the function writes something to the log file, and returns NGX\_CONF\_ERROR. That return code halts server startup. (Since the message is logged at level NGX\_LOG\_EMERG, the message will also go to standard out; FYI, [core/ngx\_log.h](http://lxr.evanmiller.org/http/source/core/ngx_log.h#L1) has a list of log levels.)

**2.4. The Module Definition**

Next we add one more layer of indirection, the ngx\_module\_t struct. The variable is called ngx\_http\_<module name>\_module. This is where references to the context and directives go, as well as the remaining callbacks (exit thread, exit process, etc.). The module definition is sometimes used as a key to look up data associated with a particular module. The module definition usually looks like this:

ngx\_module\_t ngx\_http\_<module name>\_module = {

NGX\_MODULE\_V1,

&ngx\_http\_<module name>\_module\_ctx, /\* module context \*/

ngx\_http\_<module name>\_commands, /\* module directives \*/

NGX\_HTTP\_MODULE, /\* module type \*/

NULL, /\* init master \*/

NULL, /\* init module \*/

NULL, /\* init process \*/

NULL, /\* init thread \*/

NULL, /\* exit thread \*/

NULL, /\* exit process \*/

NULL, /\* exit master \*/

NGX\_MODULE\_V1\_PADDING

};

…substituting <module name> appropriately. Modules can add callbacks for process/thread creation and death, but most modules keep things simple. (For the arguments passed to each callback, see [core/ngx\_conf\_file.h](http://lxr.evanmiller.org/http/source/core/ngx_conf_file.h#L110).)

**2.5. Module Installation**

The proper way to install a module depends on whether the module is a handler, filter, or load-balancer; so the details are reserved for those respective sections.

**3. Handlers**

Now we'll put some trivial modules under the microscope to see how they work.

**3.1. Anatomy of a Handler (Non-proxying)**

Handlers typically do four things: get the location configuration, generate an appropriate response, send the header, and send the body. A handler has one argument, the request struct. A request struct has a lot of useful information about the client request, such as the request method, URI, and headers. We'll go over these steps one by one.

**3.1.1. Getting the location configuration**

This part's easy. All you need to do is call ngx\_http\_get\_module\_loc\_conf and pass in the current request struct and the module definition. Here's the relevant part of my circle gif handler:

static ngx\_int\_t

ngx\_http\_circle\_gif\_handler(ngx\_http\_request\_t \*r)

{

ngx\_http\_circle\_gif\_loc\_conf\_t \*circle\_gif\_config;

circle\_gif\_config = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_circle\_gif\_module);

...

Now I've got access to all the variables that I set up in my merge function.

**3.1.2. Generating a response**

This is the interesting part where modules actually do work.

The request struct will be helpful here, particularly these elements:

typedef struct {

...

/\* the memory pool, used in the ngx\_palloc functions \*/

ngx\_pool\_t \*pool;

ngx\_str\_t uri;

ngx\_str\_t args;

ngx\_http\_headers\_in\_t headers\_in;

...

} ngx\_http\_request\_t;

uri is the path of the request, e.g. "/query.cgi".

args is the part of the request after the question mark (e.g. "name=john").

headers\_in has a lot of useful stuff, such as cookies and browser information, but many modules don't need anything from it. See [http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L158) if you're interested.

This should be enough information to produce some useful output. The full ngx\_http\_request\_t struct can be found in [http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L316).

**3.1.3. Sending the header**

The response headers live in a struct called headers\_out referenced by the request struct. The handler sets the ones it wants and then calls ngx\_http\_send\_header(r). Some useful parts of headers\_out include:

typedef struct {

...

ngx\_uint\_t status;

size\_t content\_type\_len;

ngx\_str\_t content\_type;

ngx\_table\_elt\_t \*content\_encoding;

off\_t content\_length\_n;

time\_t date\_time;

time\_t last\_modified\_time;

..

} ngx\_http\_headers\_out\_t;

(The rest can be found in [http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L220).)

So for example, if a module were to set the Content-Type to "image/gif", Content-Length to 100, and return a 200 OK response, this code would do the trick:

r->headers\_out.status = NGX\_HTTP\_OK;

r->headers\_out.content\_length\_n = 100;

r->headers\_out.content\_type.len = sizeof("image/gif") - 1;

r->headers\_out.content\_type.data = (u\_char \*) "image/gif";

ngx\_http\_send\_header(r);

Most legal HTTP headers are available (somewhere) for your setting pleasure. However, some headers are a bit trickier to set than the ones you see above; for example, content\_encoding has type (ngx\_table\_elt\_t\*), so the module must allocate memory for it. This is done with a function called ngx\_list\_push, which takes in an ngx\_list\_t (similar to an array) and returns a reference to a newly created member of the list (of type ngx\_table\_elt\_t). The following code sets the Content-Encoding to "deflate" and sends the header:

r->headers\_out.content\_encoding = ngx\_list\_push(&r->headers\_out.headers);

if (r->headers\_out.content\_encoding == NULL) {

return NGX\_ERROR;

}

r->headers\_out.content\_encoding->hash = 1;

r->headers\_out.content\_encoding->key.len = sizeof("Content-Encoding") - 1;

r->headers\_out.content\_encoding->key.data = (u\_char \*) "Content-Encoding";

r->headers\_out.content\_encoding->value.len = sizeof("deflate") - 1;

r->headers\_out.content\_encoding->value.data = (u\_char \*) "deflate";

ngx\_http\_send\_header(r);

This mechanism is usually used when a header can have multiple values simultaneously; it (theoretically) makes it easier for filter modules to add and delete certain values while preserving others, because they don't have to resort to string manipulation.

**3.1.4. Sending the body**

Now that the module has generated a response and put it in memory, it needs to assign the response to a special buffer, and then assign the buffer to a *chain link*, and *then* call the "send body" function on the chain link.

What are the chain links for? Nginx lets handler modules generate (and filter modules process) responses one buffer at a time; each chain link keeps a pointer to the next link in the chain, or NULL if it's the last one. We'll keep it simple and assume there is just one buffer.

First, a module will declare the buffer and the chain link:

ngx\_buf\_t \*b;

ngx\_chain\_t out;

The next step is to allocate the buffer and point our response data to it:

b = ngx\_pcalloc(r->pool, sizeof(ngx\_buf\_t));

if (b == NULL) {

ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,

"Failed to allocate response buffer.");

return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;

}

b->pos = some\_bytes; /\* first position in memory of the data \*/

b->last = some\_bytes + some\_bytes\_length; /\* last position \*/

b->memory = 1; /\* content is in read-only memory \*/

/\* (i.e., filters should copy it rather than rewrite in place) \*/

b->last\_buf = 1; /\* there will be no more buffers in the request \*/

Now the module attaches it to the chain link:

out.buf = b;

out.next = NULL;

FINALLY, we send the body, and return the status code of the output filter chain all in one go:

return ngx\_http\_output\_filter(r, &out);

Buffer chains are a critical part of Nginx's IO model, so you should be comfortable with how they work.

Trivia question: Why does the buffer have the last\_buf variable, when we can tell we're at the end of a chain by checking "next" for NULL?

Answer: A chain might be incomplete, i.e., have multiple buffers, but not all the buffers in this request or response. So some buffers are at the end of the chain but not the end of a request. This brings us to…

**3.2. Anatomy of an Upstream (a.k.a Proxy) Handler**

I waved my hands a bit about having your handler generate a response. Sometimes you'll be able to get that response just with a chunk of C code, but often you'll want to talk to another server (for example, if you're writing a module to implement another network protocol). You *could* do all of the network programming yourself, but what happens if you receive a partial response? You don't want to block the primary event loop with your own event loop while you're waiting for the rest of the response. You'd kill the Nginx's performance. Fortunately, Nginx lets you hook right into its own mechanisms for dealing with back-end servers (called "upstreams"), so your module can talk to another server without getting in the way of other requests. This section describes how a module talks to an upstream, such as Memcached, FastCGI, or another HTTP server.

**3.2.1. Summary of upstream callbacks**

Unlike the handler function for other modules, the handler function of an upstream module does little "real work". It does *not* call ngx\_http\_output\_filter. It merely sets callbacks that will be invoked when the upstream server is ready to be written to and read from. There are actually 6 available hooks:

create\_request crafts a request buffer (or chain of them) to be sent to the upstream

reinit\_request is called if the connection to the back-end is reset (just before create\_request is called for the second time)

process\_header processes the first bit of the upstream's response, and usually saves a pointer to the upstream's "payload"

abort\_request is called if the client aborts the request

finalize\_request is called when Nginx is finished reading from the upstream

input\_filter is a body filter that can be called on the response body (e.g., to remove a trailer)

How do these get attached? An example is in order. Here's a simplified version of the proxy module's handler:

static ngx\_int\_t

ngx\_http\_proxy\_handler(ngx\_http\_request\_t \*r)

{

ngx\_int\_t rc;

ngx\_http\_upstream\_t \*u;

ngx\_http\_proxy\_loc\_conf\_t \*plcf;

plcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_proxy\_module);

/\* set up our upstream struct \*/

u = ngx\_pcalloc(r->pool, sizeof(ngx\_http\_upstream\_t));

if (u == NULL) {

return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;

}

u->peer.log = r->connection->log;

u->peer.log\_error = NGX\_ERROR\_ERR;

u->output.tag = (ngx\_buf\_tag\_t) &ngx\_http\_proxy\_module;

u->conf = &plcf->upstream;

/\* attach the callback functions \*/

u->create\_request = ngx\_http\_proxy\_create\_request;

u->reinit\_request = ngx\_http\_proxy\_reinit\_request;

u->process\_header = ngx\_http\_proxy\_process\_status\_line;

u->abort\_request = ngx\_http\_proxy\_abort\_request;

u->finalize\_request = ngx\_http\_proxy\_finalize\_request;

r->upstream = u;

rc = ngx\_http\_read\_client\_request\_body(r, ngx\_http\_upstream\_init);

if (rc >= NGX\_HTTP\_SPECIAL\_RESPONSE) {

return rc;

}

return NGX\_DONE;

}

It does a bit of housekeeping, but the important parts are the callbacks. Also notice the bit about ngx\_http\_read\_client\_request\_body. That's setting another callback for when Nginx has finished reading from the client.

What will each of these callbacks do? Usually, reinit\_request, abort\_request, and finalize\_request will set or reset some sort of internal state and are only a few lines long. The real workhorses are create\_request and process\_header.

**3.2.2. The create\_request callback**

For the sake of simplicity, let's suppose I have an upstream server that reads in one character and prints out two characters. What would my functions look like?

The create\_request needs to allocate a buffer for the single-character request, allocate a chain link for that buffer, and then point the upstream struct to that chain link. It would look like this:

static ngx\_int\_t

ngx\_http\_character\_server\_create\_request(ngx\_http\_request\_t \*r)

{

/\* make a buffer and chain \*/

ngx\_buf\_t \*b;

ngx\_chain\_t \*cl;

b = ngx\_create\_temp\_buf(r->pool, sizeof("a") - 1);

if (b == NULL)

return NGX\_ERROR;

cl = ngx\_alloc\_chain\_link(r->pool);

if (cl == NULL)

return NGX\_ERROR;

/\* hook the buffer to the chain \*/

cl->buf = b;

/\* chain to the upstream \*/

r->upstream->request\_bufs = cl;

/\* now write to the buffer \*/

b->pos = "a";

b->last = b->pos + sizeof("a") - 1;

return NGX\_OK;

}

That wasn't so bad, was it? Of course, in reality you'll probably want to use the request URI in some meaningful way. It's available as an ngx\_str\_t in r->uri, and the GET paramaters are in r->args, and don't forget you also have access to the request headers and cookies.

**3.2.3. The process\_header callback**

Now it's time for the process\_header. Just as create\_request added a pointer to the request body, process\_header*shifts the response pointer to the part that the client will receive*. It also reads in the header from the upstream and sets the client response headers accordingly.

Here's a bare-minimum example, reading in that two-character response. Let's suppose the first character is the "status" character. If it's a question mark, we want to return a 404 File Not Found to the client and disregard the other character. If it's a space, then we want to return the other character to the client along with a 200 OK response. All right, it's not the most useful protocol, but it's a good demonstration. How would we write this process\_header function?

static ngx\_int\_t

ngx\_http\_character\_server\_process\_header(ngx\_http\_request\_t \*r)

{

ngx\_http\_upstream\_t \*u;

u = r->upstream;

/\* read the first character \*/

switch(u->buffer.pos[0]) {

case '?':

r->header\_only; /\* suppress this buffer from the client \*/

u->headers\_in.status\_n = 404;

break;

case ' ':

u->buffer.pos++; /\* move the buffer to point to the next character \*/

u->headers\_in.status\_n = 200;

break;

}

return NGX\_OK;

}

That's it. Manipulate the header, change the pointer, it's done. Notice that headers\_in is actually a response header struct like we've seen before (cf. [http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L158)), but it can be populated with the headers from the upstream. A real proxying module will do a lot more header processing, not to mention error handling, but you get the main idea.

But… what if we don't have the whole header from the upstream in one buffer?

**3.2.4. Keeping state**

Well, remember how I said that abort\_request, reinit\_request, and finalize\_request could be used for resetting internal state? That's because many upstream modules *have* internal state. The module will need to define a *custom context struct* to keep track of what it has read so far from an upstream. This is NOT the same as the "Module Context" referred to above. That's of a pre-defined type, whereas the custom context can have whatever elements and data you need (it's your struct). This context struct should be instantiated inside the create\_request function, perhaps like this:

ngx\_http\_character\_server\_ctx\_t \*p; /\* my custom context struct \*/

p = ngx\_pcalloc(r->pool, sizeof(ngx\_http\_character\_server\_ctx\_t));

if (p == NULL) {

return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;

}

ngx\_http\_set\_ctx(r, p, ngx\_http\_character\_server\_module);

That last line essentially registers the custom context struct with a particular request and module name for easy retrieval later. Whenever you need this context struct (probably in all the other callbacks), just do:

ngx\_http\_proxy\_ctx\_t \*p;

p = ngx\_http\_get\_module\_ctx(r, ngx\_http\_proxy\_module);

And p will have the current state. Set it, reset it, increment, decrement, shove arbitrary data in there, whatever you want. This is a great way to use a persistent state machine when reading from an upstream that returns data in chunks, again without blocking the primary event loop. Nice!

**3.3. Handler Installation**

Handlers are installed by adding code to the callback of the directive that enables the module. For example, my circle gif ngx\_command\_t looks like this:

{ ngx\_string("circle\_gif"),

NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_NOARGS,

ngx\_http\_circle\_gif,

0,

0,

NULL }

The callback is the third element, in this case ngx\_http\_circle\_gif. Recall that the arguments to this callback are the directive struct (ngx\_conf\_t, which holds the user's arguments), the relevant ngx\_command\_t struct, and a pointer to the module's custom configuration struct. For my circle gif module, the function looks like:

static char \*

ngx\_http\_circle\_gif(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf)

{

ngx\_http\_core\_loc\_conf\_t \*clcf;

clcf = ngx\_http\_conf\_get\_module\_loc\_conf(cf, ngx\_http\_core\_module);

clcf->handler = ngx\_http\_circle\_gif\_handler;

return NGX\_CONF\_OK;

}

There are two steps here: first, get the "core" struct for this location, then assign a handler to it. Pretty simple, eh?

I've said all I know about handler modules. It's time to move onto filter modules, the components in the output filter chain.

**4. Filters**

Filters manipulate responses generated by handlers. Header filters manipulate the HTTP headers, and body filters manipulate the response content.

**4.1. Anatomy of a Header Filter**

A header filter consists of three basic steps:

1. Decide whether to operate on this response
2. Operate on the response
3. Call the next filter

To take an example, here's a simplified version of the "not modified" header filter, which sets the status to 304 Not Modified if the client's If-Modified-Since header matches the response's Last-Modified header. Note that header filters take in the ngx\_http\_request\_t struct as the only argument, which gets us access to both the client headers and soon-to-be-sent response headers.

static

ngx\_int\_t ngx\_http\_not\_modified\_header\_filter(ngx\_http\_request\_t \*r)

{

time\_t if\_modified\_since;

if\_modified\_since = ngx\_http\_parse\_time(r->headers\_in.if\_modified\_since->value.data,

r->headers\_in.if\_modified\_since->value.len);

/\* step 1: decide whether to operate \*/

if (if\_modified\_since != NGX\_ERROR &&

if\_modified\_since == r->headers\_out.last\_modified\_time) {

/\* step 2: operate on the header \*/

r->headers\_out.status = NGX\_HTTP\_NOT\_MODIFIED;

r->headers\_out.content\_type.len = 0;

ngx\_http\_clear\_content\_length(r);

ngx\_http\_clear\_accept\_ranges(r);

}

/\* step 3: call the next filter \*/

return ngx\_http\_next\_header\_filter(r);

}

The headers\_out structure is just the same as we saw in the section about handlers (cf. [http/ngx\_http\_request.h](http://lxr.evanmiller.org/http/source/http/ngx_http_request.h#L220)), and can be manipulated to no end.

**4.2. Anatomy of a Body Filter**

The buffer chain makes it a little tricky to write a body filter, because the body filter can only operate on one buffer (chain link) at a time. The module must decide whether to *overwrite* the input buffer, *replace* the buffer with a newly allocated buffer, or *insert* a new buffer before or after the buffer in question. To complicate things, sometimes a module will receive several buffers so that it has an *incomplete buffer chain*that it must operate on. Unfortunately, Nginx does not provide a high-level API for manipulating the buffer chain, so body filters can be difficult to understand (and to write). But, here are some operations you might see in action.

A body filter's prototype might look like this (example taken from the "chunked" filter in the Nginx source):

static ngx\_int\_t ngx\_http\_chunked\_body\_filter(ngx\_http\_request\_t \*r, ngx\_chain\_t \*in);

The first argument is our old friend the request struct. The second argument is a pointer to the head of the current partial chain (which could contain 0, 1, or more buffers).

Let's take a simple example. Suppose we want to insert the text "<l!-- Served by Nginx -->" to the end of every request. First, we need to figure out if the response's final buffer is included in the buffer chain we were given. Like I said, there's not a fancy API, so we'll be rolling our own for loop:

ngx\_chain\_t \*chain\_link;

int chain\_contains\_last\_buffer = 0;

chain\_link = in;

for ( ; ; ) {

if (chain\_link->buf->last\_buf)

chain\_contains\_last\_buffer = 1;

if (chain\_link->next == NULL)

break;

chain\_link = chain\_link->next;

}

Now let's bail out if we don't have that last buffer:

if (!chain\_contains\_last\_buffer)

return ngx\_http\_next\_body\_filter(r, in);

Super, now the last buffer is stored in chain\_link. Now we allocate a new buffer:

ngx\_buf\_t \*b;

b = ngx\_calloc\_buf(r->pool);

if (b == NULL) {

return NGX\_ERROR;

}

And put some data in it:

b->pos = (u\_char \*) "<!-- Served by Nginx -->";

b->last = b->pos + sizeof("<!-- Served by Nginx -->") - 1;

And hook the buffer into a new chain link:

ngx\_chain\_t \*added\_link;

added\_link = ngx\_alloc\_chain\_link(r->pool);

if (added\_link == NULL)

return NGX\_ERROR;

added\_link->buf = b;

added\_link->next = NULL;

Finally, hook the new chain link to the final chain link we found before:

chain\_link->next = added\_link;

And reset the "last\_buf" variables to reflect reality:

chain\_link->buf->last\_buf = 0;

added\_link->buf->last\_buf = 1;

And pass along the modified chain to the next output filter:

return ngx\_http\_next\_body\_filter(r, in);

The resulting function takes much more effort than what you'd do with, say, mod\_perl ($response->body =~ s/$/<!-- Served by mod\_perl -->/), but the buffer chain is a very powerful construct, allowing programmers to process data incrementally so that the client gets something as soon as possible. However, in my opinion, the buffer chain desperately needs a cleaner interface so that programmers can't leave the chain in an inconsistent state. For now, manipulate it at your own risk.

**4.3. Filter Installation**

Filters are installed in the post-configuration step. We install both header filters and body filters in the same place.

Let's take a look at the chunked filter module for a simple example. Its module context looks like this:

static ngx\_http\_module\_t ngx\_http\_chunked\_filter\_module\_ctx = {

NULL, /\* preconfiguration \*/

ngx\_http\_chunked\_filter\_init, /\* postconfiguration \*/

...

};

Here's what happens in ngx\_http\_chunked\_filter\_init:

static ngx\_int\_t

ngx\_http\_chunked\_filter\_init(ngx\_conf\_t \*cf)

{

ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;

ngx\_http\_top\_header\_filter = ngx\_http\_chunked\_header\_filter;

ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;

ngx\_http\_top\_body\_filter = ngx\_http\_chunked\_body\_filter;

return NGX\_OK;

}

What's going on here? Well, if you remember, filters are set up with a CHAIN OF RESPONSIBILITY. When a handler generates a response, it calls two functions: ngx\_http\_output\_filter, which calls the global function reference ngx\_http\_top\_body\_filter; and ngx\_http\_send\_header, which calls the global function reference ngx\_http\_top\_header\_filter.

ngx\_http\_top\_body\_filter and ngx\_http\_top\_header\_filter are the respective "heads" of the body and header filter chains. Each "link" on the chain keeps a function reference to the next link in the chain (the references are called ngx\_http\_next\_body\_filter and ngx\_http\_next\_header\_filter). When a filter is finished executing, it just calls the next filter, until a specially defined "write" filter is called, which wraps up the HTTP response. What you see in this filter\_init function is the module adding itself to the filter chains; it keeps a reference to the old "top" filters in its own "next" variables and declares *its* functions to be the new "top" filters. (Thus, the last filter to be installed is the first to be executed.)

Side note: how does this work exactly?

Each filter either returns an error code or uses this as the return statement:

return ngx\_http\_next\_body\_filter();

Thus, if the filter chain reaches the (specially-defined) end of the chain, an "OK" response is returned, but if there's an error along the way, the chain is cut short and Nginx serves up the appropriate error message. It's a singly-linked list with fast failures implemented solely with function references. Brilliant.

**5. Load-Balancers**

A load-balancer is just a way to decide which backend server will receive a particular request; implementations exist for distributing requests in round-robin fashion or hashing some information about the request. This section will describe both a load-balancer's installation and its invocation, using the upstream\_hash module ([full source](http://www.evanmiller.org/nginx/ngx_http_upstream_hash_module.c.txt)) as an example. upstream\_hash chooses a backend by hashing a variable specified in nginx.conf.

A load-balancing module has six pieces:

1. The enabling configuration directive (e.g, hash;) will call a *registration function*
2. The registration function will define the legal server options (e.g., weight=) and register an *upstream initialization function*
3. The upstream initialization function is called just after the configuration is validated, and it:
   * resolves the server names to particular IP addresses
   * allocates space for sockets
   * sets a callback to the *peer initialization function*
4. the peer initialization function, called once per request, sets up data structures that the *load-balancing function* will access and manipulate;
5. the load-balancing function decides where to route requests; it is called at least once per client request (more, if a backend request fails). This is where the interesting stuff happens.
6. and finally, the *peer release function* can update statistics after communication with a particular backend server has finished (whether successfully or not)

It's a lot, but I'll break it down into pieces.

**5.1. The enabling directive**

Directive declarations, recall, specify both where they're valid and a function to call when they're encountered. A directive for a load-balancer should have the NGX\_HTTP\_UPS\_CONF flag set, so that Nginx knows this directive is only valid inside an upstream block. It should provide a pointer to a *registration function*. Here's the directive declaration from the upstream\_hash module:

{ ngx\_string("hash"),

NGX\_HTTP\_UPS\_CONF|NGX\_CONF\_NOARGS,

ngx\_http\_upstream\_hash,

0,

0,

NULL },

Nothing new there.

**5.2. The registration function**

The callback ngx\_http\_upstream\_hash above is the registration function, so named (by me) because it registers an *upstream initialization function* with the surrounding upstream configuration. In addition, the registration function defines which options to the server directive are legal inside this particular upstreamblock (e.g., weight=, fail\_timeout=). Here's the registration function of the upstream\_hash module:

ngx\_http\_upstream\_hash(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf)

{

ngx\_http\_upstream\_srv\_conf\_t \*uscf;

ngx\_http\_script\_compile\_t sc;

ngx\_str\_t \*value;

ngx\_array\_t \*vars\_lengths, \*vars\_values;

value = cf->args->elts;

/\* the following is necessary to evaluate the argument to "hash" as a $variable \*/

ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));

vars\_lengths = NULL;

vars\_values = NULL;

sc.cf = cf;

sc.source = &value[1];

sc.lengths = &vars\_lengths;

sc.values = &vars\_values;

sc.complete\_lengths = 1;

sc.complete\_values = 1;

if (ngx\_http\_script\_compile(&sc) != NGX\_OK) {

return NGX\_CONF\_ERROR;

}

/\* end of $variable stuff \*/

uscf = ngx\_http\_conf\_get\_module\_srv\_conf(cf, ngx\_http\_upstream\_module);

/\* the upstream initialization function \*/

uscf->peer.init\_upstream = ngx\_http\_upstream\_init\_hash;

uscf->flags = NGX\_HTTP\_UPSTREAM\_CREATE;

/\* OK, more $variable stuff \*/

uscf->values = vars\_values->elts;

uscf->lengths = vars\_lengths->elts;

/\* set a default value for "hash\_method" \*/

if (uscf->hash\_function == NULL) {

uscf->hash\_function = ngx\_hash\_key;

}

return NGX\_CONF\_OK;

}

Aside from jumping through hoops so we can evaluation $variable later, it's pretty straightforward; assign a callback, set some flags. What flags are available?

* NGX\_HTTP\_UPSTREAM\_CREATE: let there be server directives in this upstream block. I can't think of a situation where you wouldn't use this.
* NGX\_HTTP\_UPSTREAM\_WEIGHT: let the server directives take a weight= option
* NGX\_HTTP\_UPSTREAM\_MAX\_FAILS: allow the max\_fails= option
* NGX\_HTTP\_UPSTREAM\_FAIL\_TIMEOUT: allow the fail\_timeout= option
* NGX\_HTTP\_UPSTREAM\_DOWN: allow the down option
* NGX\_HTTP\_UPSTREAM\_BACKUP: allow the backup option

Each module will have access to these configuration values. *It's up to the module to decide what to do with them.* That is, max\_fails will not be automatically enforced; all the failure logic is up to the module author. More on that later. For now, we still haven't finished followed the trail of callbacks. Next up, we have the upstream initialization function (the init\_upstream callback in the previous function).

**5.3. The upstream initialization function**

The purpose of the upstream initialization function is to resolve the host names, allocate space for sockets, and assign (yet another) callback. Here's how upstream\_hash does it:

ngx\_int\_t

ngx\_http\_upstream\_init\_hash(ngx\_conf\_t \*cf, ngx\_http\_upstream\_srv\_conf\_t \*us)

{

ngx\_uint\_t i, j, n;

ngx\_http\_upstream\_server\_t \*server;

ngx\_http\_upstream\_hash\_peers\_t \*peers;

/\* set the callback \*/

us->peer.init = ngx\_http\_upstream\_init\_upstream\_hash\_peer;

if (!us->servers) {

return NGX\_ERROR;

}

server = us->servers->elts;

/\* figure out how many IP addresses are in this upstream block. \*/

/\* remember a domain name can resolve to multiple IP addresses. \*/

for (n = 0, i = 0; i < us->servers->nelts; i++) {

n += server[i].naddrs;

}

/\* allocate space for sockets, etc \*/

peers = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_upstream\_hash\_peers\_t)

+ sizeof(ngx\_peer\_addr\_t) \* (n - 1));

if (peers == NULL) {

return NGX\_ERROR;

}

peers->number = n;

/\* one port/IP address per peer \*/

for (n = 0, i = 0; i < us->servers->nelts; i++) {

for (j = 0; j < server[i].naddrs; j++, n++) {

peers->peer[n].sockaddr = server[i].addrs[j].sockaddr;

peers->peer[n].socklen = server[i].addrs[j].socklen;

peers->peer[n].name = server[i].addrs[j].name;

}

}

/\* save a pointer to our peers for later \*/

us->peer.data = peers;

return NGX\_OK;

}

This function is a bit more involved than one might hope. Most of the work seems like it should be abstracted, but it's not, so that's what we live with. One strategy for simplifying things is to call the upstream initialization function of another module, have it do all the dirty work (peer allocation, etc), and then override the us->peer.init callback afterwards. For an example, see [http/modules/ngx\_http\_upstream\_ip\_hash\_module.c](http://lxr.evanmiller.org/http/source/http/modules/ngx_http_upstream_ip_hash_module.c#L80).

The important bit from our point of view is setting a pointer to the *peer initialization function*, in this case ngx\_http\_upstream\_init\_upstream\_hash\_peer.

**5.4. The peer initialization function**

The peer initialization function is called once per request. It sets up a data structure that the module will use as it tries to find an appropriate backend server to service that request; this structure is persistent across backend re-tries, so it's a convenient place to keep track of the number of connection failures, or a computed hash value. By convention, this struct is called ngx\_http\_upstream\_<module name>\_peer\_data\_t.

In addition, the peer initalization function sets up two callbacks:

* get: the load-balancing function
* free: the peer release function (usually just updates some statistics when a connection finishes)

As if that weren't enough, it also initalizes a variable called tries. As long as tries is positive, nginx will keep retrying this load-balancer. When tries is zero, nginx will give up. It's up to the get and free functions to set tries appropriately.

Here's a peer initialization function from the upstream\_hash module:

static ngx\_int\_t

ngx\_http\_upstream\_init\_hash\_peer(ngx\_http\_request\_t \*r,

ngx\_http\_upstream\_srv\_conf\_t \*us)

{

ngx\_http\_upstream\_hash\_peer\_data\_t \*uhpd;

ngx\_str\_t val;

/\* evaluate the argument to "hash" \*/

if (ngx\_http\_script\_run(r, &val, us->lengths, 0, us->values) == NULL) {

return NGX\_ERROR;

}

/\* data persistent through the request \*/

uhpd = ngx\_pcalloc(r->pool, sizeof(ngx\_http\_upstream\_hash\_peer\_data\_t)

+ sizeof(uintptr\_t)

\* ((ngx\_http\_upstream\_hash\_peers\_t \*)us->peer.data)->number

/ (8 \* sizeof(uintptr\_t)));

if (uhpd == NULL) {

return NGX\_ERROR;

}

/\* save our struct for later \*/

r->upstream->peer.data = uhpd;

uhpd->peers = us->peer.data;

/\* set the callbacks and initialize "tries" to "hash\_again" + 1\*/

r->upstream->peer.free = ngx\_http\_upstream\_free\_hash\_peer;

r->upstream->peer.get = ngx\_http\_upstream\_get\_hash\_peer;

r->upstream->peer.tries = us->retries + 1;

/\* do the hash and save the result \*/

uhpd->hash = us->hash\_function(val.data, val.len);

return NGX\_OK;

}

That wasn't so bad. Now we're ready to pick an upstream server.

**5.5. The load-balancing function**

It's time for the main course. The real meat and potatoes. This is where the module picks an upstream. The load-balancing function's prototype looks like:

static ngx\_int\_t

ngx\_http\_upstream\_get\_<module\_name>\_peer(ngx\_peer\_connection\_t \*pc, void \*data);

data is our struct of useful information concerning this client connection. pc will have information about the server we're going to connect to. The job of the load-balancing function is to fill in values for pc->sockaddr, pc->socklen, and pc->name. If you know some network programming, then those variable names might be familiar; but they're actually not very important to the task at hand. We don't care what they stand for; we just want to know where to find appropriate values to fill them.

This function must find a list of available servers, choose one, and assign its values to pc. Let's look at how upstream\_hash does it.

upstream\_hash previously stashed the server list into the ngx\_http\_upstream\_hash\_peer\_data\_t struct back in the call to ngx\_http\_upstream\_init\_hash (above). This struct is now available as data:

ngx\_http\_upstream\_hash\_peer\_data\_t \*uhpd = data;

The list of peers is now stored in uhpd->peers->peer. Let's pick a peer from this array by dividing the computed hash value by the number of servers:

ngx\_peer\_addr\_t \*peer = &uhpd->peers->peer[uhpd->hash % uhpd->peers->number];

Now for the grand finale:

pc->sockaddr = peer->sockaddr;

pc->socklen = peer->socklen;

pc->name = &peer->name;

return NGX\_OK;

That's all! If the load-balancer returns NGX\_OK, it means, "go ahead and try this server". If it returns NGX\_BUSY, it means all the backend hosts are unavailable, and Nginx should try again.

But… how do we keep track of what's unavailable? And what if we don't want it to try again?

**5.6. The peer release function**

The peer release function operates after an upstream connection takes place; its purpose is to track failures. Here is its function prototype:

void

ngx\_http\_upstream\_free\_<module name>\_peer(ngx\_peer\_connection\_t \*pc, void \*data,

ngx\_uint\_t state);

The first two parameters are just the same as we saw in the load-balancer function. The third parameter is a state variable, which indicates whether the connection was successful. It may contain two values bitwise OR'd together: NGX\_PEER\_FAILED (the connection failed) and NGX\_PEER\_NEXT (either the connection failed, or it succeeded but the application returned an error). Zero means the connection succeeded.

It's up to the module author to decide what to do about these failure events. If they are to be used at all, the results should be stored in data, a pointer to the custom per-request data struct.

But the crucial purpose of the peer release function is to set pc->tries to zero if you don't want Nginx to keep trying this load-balancer during this request. The simplest peer release function would look like this:

pc->tries = 0;

That would ensure that if there's ever an error reaching a backend server, a 502 Bad Proxy error will be returned to the client.

Here's a more complicated example, taken from the upstream\_hash module. If a backend connection fails, it marks it as failed in a bit-vector (called tried, an array of type uintptr\_t), then keeps choosing a new backend until it finds one that has not failed.

#define ngx\_bitvector\_index(index) index / (8 \* sizeof(uintptr\_t))

#define ngx\_bitvector\_bit(index) (uintptr\_t) 1 << index % (8 \* sizeof(uintptr\_t))

static void

ngx\_http\_upstream\_free\_hash\_peer(ngx\_peer\_connection\_t \*pc, void \*data,

ngx\_uint\_t state)

{

ngx\_http\_upstream\_hash\_peer\_data\_t \*uhpd = data;

ngx\_uint\_t current;

if (state & NGX\_PEER\_FAILED

&& --pc->tries)

{

/\* the backend that failed \*/

current = uhpd->hash % uhpd->peers->number;

/\* mark it in the bit-vector \*/

uhpd->tried[ngx\_bitvector\_index(current)] |= ngx\_bitvector\_bit(current);

do { /\* rehash until we're out of retries or we find one that hasn't been tried \*/

uhpd->hash = ngx\_hash\_key((u\_char \*)&uhpd->hash, sizeof(ngx\_uint\_t));

current = uhpd->hash % uhpd->peers->number;

} while ((uhpd->tried[ngx\_bitvector\_index(current)] & ngx\_bitvector\_bit(current)) && --pc->tries);

}

}

This works because the load-balancing function will just look at the new value of uhpd->hash.

Many applications won't need retry or high-availability logic, but it's possible to provide it with just a few lines of code like you see here.

**6. Writing and Compiling a New Nginx Module**

So by now, you should be prepared to look at an Nginx module and try to understand what's going on (and you'll know where to look for help). Take a look in [src/http/modules/](http://lxr.evanmiller.org/http/source/http/modules/) to see the available modules. Pick a module that's similar to what you're trying to accomplish and look through it. Stuff look familiar? It should. Refer between this guide and the module source to get an understanding about what's going on.

But Emiller didn't write a *Balls-In Guide to Reading Nginx Modules*. Hell no. This is a *Balls-Out Guide*. We're not reading. We're writing. Creating. Sharing with the world.

First thing, you're going to need a place to work on your module. Make a folder for your module anywhere on your hard drive, but separate from the Nginx source (and make sure you have the latest copy from [nginx.net](http://nginx.net/)). Your new folder should contain two files to start with:

* "config"
* "ngx\_http\_<your module>\_module.c"

The "config" file will be included by ./configure, and its contents will depend on the type of module.

**"config" for filter modules:**

ngx\_addon\_name=ngx\_http\_<your module>\_module

HTTP\_AUX\_FILTER\_MODULES="$HTTP\_AUX\_FILTER\_MODULES ngx\_http\_<your module>\_module"

NGX\_ADDON\_SRCS="$NGX\_ADDON\_SRCS $ngx\_addon\_dir/ngx\_http\_<your module>\_module.c"

**"config" for other modules:**

ngx\_addon\_name=ngx\_http\_<your module>\_module

HTTP\_MODULES="$HTTP\_MODULES ngx\_http\_<your module>\_module"

NGX\_ADDON\_SRCS="$NGX\_ADDON\_SRCS $ngx\_addon\_dir/ngx\_http\_<your module>\_module.c"

Now for your C file. I recommend copying an existing module that does something similar to what you want, but rename it "ngx\_http\_<your module>\_module.c". Let this be your model as you change the behavior to suit your needs, and refer to this guide as you understand and refashion the different pieces.

When you're ready to compile, just go into the Nginx directory and type

./configure --add-module=path/to/your/new/module/directory

and then make and make install like you normally would. If all goes well, your module will be compiled right in. Nice, huh? No need to muck with the Nginx source, and adding your module to new versions of Nginx is a snap, just use that same ./configure command. By the way, if your module needs any dynamically linked libraries, you can add this to your "config" file:

CORE\_LIBS="$CORE\_LIBS -lfoo"

Where foo is the library you need. If you make a cool or useful module, be sure to send a note to the [Nginx mailing list](http://wiki.codemongers.com/MailinglistSubscribe) and share your work.

**7. Advanced Topics**

This guide covers the basics of Nginx module development. For tips on writing more sophisticated modules, be sure to check out [*Emiller's Advanced Topics In Nginx Module Development*](http://www.evanmiller.org/nginx-modules-guide-advanced.html).

**Appendix A: Code References**

[Nginx source tree (cross-referenced)](http://lxr.evanmiller.org/http/source/)

[Nginx module directory (cross-referenced)](http://lxr.evanmiller.org/http/source/http/modules/)

[Example addon: circle\_gif](http://www.evanmiller.org/nginx/ngx_http_circle_gif_module.c.txt)

[Example addon: upstream\_hash](http://www.evanmiller.org/nginx/ngx_http_upstream_hash_module.c.txt)

[Example addon: upstream\_fair](http://github.com/gnosek/nginx-upstream-fair/tree/master)

**Appendix B: Changelog**

* January 16, 2013: Corrected code sample in 5.5.
* December 20, 2011: Corrected code sample in 4.2 (one more time).
* March 14, 2011: Corrected code sample in 4.2 (again).
* November 11, 2009: Corrected code sample in 4.2.
* August 13, 2009: Reorganized, and moved [*Advanced Topics*](http://www.evanmiller.org/nginx-modules-guide-advanced.html) to a separate article.
* July 23, 2009: Corrected code sample in 3.5.3.
* December 24, 2008: Corrected code sample in 3.4.
* July 14, 2008: Added information about subrequests; slight reorganization
* July 12, 2008: Added [Grzegorz Nosek](http://localdomain.pl/)'s guide to shared memory
* July 2, 2008: Corrected "config" file for filter modules; rewrote introduction; added TODO section
* May 28, 2007: Changed the load-balancing example to the simpler upstream\_hash module
* May 19, 2007: Corrected bug in body filter example
* May 4, 2007: Added information about load-balancers
* April 28, 2007: Initial draft