Table of Contents

Table of Contents

Introduction

A brief introduction to programming roxen; glossary and concepts.

End of /roxen/2.1/programmer/introduction/index.xml

Terminology

Start of /roxen/2.1/programmer/introduction/terminology.xml

Some important terms to grasp to fully understand the manual.

- A component of a virtual server that fills some specific purpose, such as RXML parsing, providing a filesystem or similar. A roxen module is a piece of code interfacing to the roxen module API, extending the server's features. Roxen modules may be implemented in pike or Java.
- A pike module is, unlike roxen modules, not something seen by the common roxen user or administrator. A pike module extends the pike language environment, providing the programmer alone with some features, abstraction or similar, such as the SQL and Image modules.
- The virtual filesystem is the entire path namespace of your server, as seen in the URL's path segment. In a basic roxen setup with just a common filesystem module mounted at `/` (the filesystem root), all files in the virtual filesystem map to this one filesystem, as is typically the case with other webservers where you have a single document root.

  Roxen, however, has a slightly different approach to filesystems, that shares many characteristics with Unix filesystems. Several filesystems may be mounted on top of each other at different, or indeed the same, mountpoints in the virtual namespace of the server. If one filesystem is mounted at `/`, another at `/I/live/` and a third at `/I/live/here/`, a request for the file `/I/live/here/happily.html` may deliver a file residing in either filesystem. Which one is decided first by the longest matching mountpoint, secondly the filesystem modules' priorities (as set by the administrator) and finally by which filesystem actually has a file to deliver.

  Similarly, even entire other virtual servers may be mounted somewhere within your server's namespace, effectively blacking it out to your server. This also means that such a server's virtual filesystem does not map one-to-one to the path namespace as seen in the URL from the browser's point of view. What to the browser looks like a request of the file `/my/secret/garden.jpg` could quite possibly be the file `/secret/garden.jpg` from a virtual server mounted at `/my/`.

  Due to the dynamic nature of roxen, a file in the virtual filesystem need not exist as an actual file on disk (as it would in a real filesystem), but might be the result of some on-the-fly file-generation activity in a roxen module of some sort. Indeed, even the error page received when roxen didn't find a file, exists in the virtual filesystem (wherever anything else doesn't).

  To find out what actually happens when requesting a given URL from your server; where the file has its origin,

what modules touch it and so on, the "Resolve path..." option in the Maintenance menu under the Tasks tab in the administration interface does wonders.

- The real filesystem refers, unlike the virtual filesystem discussed above, to the filesystems provided by the operating system, where actual files reside; typically on a hard disk, network filesystem or similar. As the name hints, files on disk might be considered just the slightest bit more real than the figments of imagination that may be harvested from the virtual filesystem.
- The term refers to where location modules are found in the virtual filesystem. A virtual file's path is considered to be below a certain mountpoint when the path begins with the the entire length of the mountpoint; for instance, `/demo/4` is below both mountpoints `/` and `/demo/` but not below `/demo/module/` or `/hello/world/`. A mount point need not necessarily end in a slash, but it is a rather common convention.
- Back in the old days, a web server was one website serving files through one port at one host. Things change, and the world grows more complex. The widespread trend of abstracting a concept and tagging it "virtual" applies to virtual servers, as much as it does to the virtual filesystem; a virtual server in roxen is a collection of roxen modules working together under a common roof.

  The criterion of being accessible on a single host:port combination does not apply in roxen, nor the fact that the server uses the entire namespace of that domain. By using several server URL:s for a virtual server, it can respond to requests for several domains, on several ports possibly using several different protocols (eg HTTP, HTTPS and FTP, all to the same virtual filesystem). Or, just as possible, if given the sole server URL `http://my.domain/just/here/`, it will not see requests for any other part of the namespace of some other server handling `http://my.domain/` than those below `/just/here/`. Equally possible, although not the most common of configurations, is not to mount any port at all.

End of /roxen/2.1/programmer/introduction/terminology.xml

The Inner Workings of Roxen

Start of /roxen/2.1/programmer/introduction/base_concepts.xml

In programming roxen, it is helpful to be aware of how the larger building blocks fit together, what goes where and so on to have a fair grasp of the general layout of the programming environment. This section will rush through roxen from one end to the other, noting important and/or useful concepts in the process. Understanding everything along the way isn't essential, but having once heard about them doesn't hurt.

Roxen is a collection of pike scripts and modules. When firing up the start script, it sets up some essential environment variables to keep dynamic libraries and various other external stuff happy. After a rapid progression of start script argument parsing

and setting up paths, it rotates the logs and launches the first pikescript; `server/base_server/roxenloader.pike`. Standard output and the standard error streams, stderr for short, is redirected to the debug log, unless you launched the start script with the `--once` parameter, in which case you'll see everything in your shell console. We mention this, since it's very handy while developing.

Before racing off the topic of the start script, there is at least one other set of options essential to the developer, namely defines. Hidden all over roxen to the casual user are some hefty amounts of #ifdef:ed helpful debug code that would swamp the logs if always turned on. To set a define, append -Ddefine_name to the command line, as in -DREQUEST_DEBUG (for some rather verbose information about most things happening with each request as it passes through roxen).

Next in turn, the roxenloader bootstraps the server, making sure some dependencies are met, setting up some constants not commonly present in pike and installs roxen's own pike master program, `server/etc/roxen_master.pike`. This program is responsible for, among other things, the dumping and reloading of programs.

Then, the very core of roxen, `server/base_server/ roxen.pike` gets loaded. For generally speedy loading, roxen dumps a lot of its components once compiled, to be loaded back instantaneously next time if started with the same set of defines. The configuration of all virtual servers are loaded next, their respective ports get registered and unless the flag `--no-delayed-load` was passed to the start script, not much more happens next until the first request arrives to a server. This means, of course, this code in a module of yours won't run until a server using the module is needed.

At this stage, the server process is up and about and will listen to incoming requests on all registered ports. When a request to an uninitialized server is received, roxen will initialize that server, loading and strapping all of its modules. Once done, the request is sent forth through the common processing sequence. Your applications will not be affected by the delayed loading, but nevertheless it is healthy knowing what happens and how. To the application, whether it be a module, a script, an rxml page or a servlet, the delayed loading is transparent and does not interfere with the programming environment.

For the rest of the lifespan of the roxen process, it will go about its business listening and responding to requests, until terminated with a signal to the process, the start script or by the restart/shutdown administrator action. When the roxen process receives a SIGHUP signal, it reloads its configuration files. A SIGINT/SIGTERM signal takes down the process and makes the start script spawn off a new one to replace the old one (sending a SIGINT/SIGTERM to the start script shuts down the server without respawning another one).

End of /roxen/2.1/programmer/introduction/ base_concepts.xml

A Request's Path Through Roxen

A chapter devoted to how a request is processed/routed through the various layers inside roxen, lightly addressing the module type calling sequence, but mostly intended to give a sound perspective of what happens where and a fair grasp of how roxen works inside. This chapter is explicitly not limited to pike programming, in that it also outlines where CGI, java servlets, perl, php et cetera fits in the overall picture.

End of /roxen/2.1/programmer/request/index.xml

Encountered Module Types

Start of /roxen/2.1/programmer/request/path.xml

This page follows a request's path from browser to server, through the various layers of roxen, the module types and to the end of the chain where the resulting document is sent back again to the browser as the reply to the request. After reading this, you should have a good grasp on what happens when and how the various module types fit in the overall picture.

For this example walk-through, we will follow a request from browser through the server and back; the example applies for all supported protocols. We'll be abstracting all encountered modules by type instead of by name, since we're getting at the principle rather than the specifics of a specific server configuration.

Calling Sequence

This is a description of the order of calling modules in roxen. Generally speaking, an incoming request passes through a number of *type levels*, which will be described in turn. A failure at a type level means that none of the modules of that type could treat the request. The case where there are no modules of a certain type is a trivial case of failure.

A failure usually means that the request is passed on to the next level. What happens when a module succeeds in treating the request depends on the level, and will be described later for each case in the roxen modules chapter.

1. An incoming request enters roxen through the protocol module, which handles the lower level communication with the client.

2. If the protocol module got some form of authentication information from the client, the authentication module is invoked. Regardless of the success or failure, the request moves on to the next layer. The authentication status (fail or a valid user identity) is stored in the request information object.

3. The first try modules get the first shot at returning a response of some sort to the client. From here on, success or failure means breaking out of or staying with the flow of the calling sequence; handled requests are sent back to the client, unhandled are subject to enter the other module types further down the chain.

4. The request now enters a location module; which one depends on the path accessed. In this respect, the location modules work almost like your average file system; a given path refers to a certain file entry on some storage medium somewhere. Or, possibly, a directory entry or a non-existent file. In either of the latter cases, the request moves on.

5. The request was found a directory at some earlier level, and it is now up to the directory listing module to generate some form of directory listing or representation of the directory at hand.

6. If some previous level sent handled a request by sending forth a file down the chain, it is processed by an appropriate file extension module (if one handling the proper extension was available).

7. The content-type module tags the resulting page with a suitable content-type for the file being sent back to the browser. Modules may of course override this, should they know what they want.

8. All requests then pass through the filter module stage. Filter modules may process and alter the request at leisure, watermarking, filtering out information or doing other forms of post processing.

9. If no module has yet handled the request, the last modules get a shot at catching and processing the request before a file not found error is sent back to the client.

10. The protocol module which originally set this chain going is returned the result from previous stages and starts sending the result to the client in response to the request.

11. Finally, as the result is being transferred back to the client, the logger modules get their peek at the request. When the logger modules are done and the whole response is sent to the browser, the request information object dies and the request is over.

End of /roxen/2.1/programmer/request/path.xml

Important Concepts

A chapter on various concepts that are of some importance from a programmer's point of view, but that did not fill entire chapters of their own.

End of /roxen/2.1/programmer/concepts/index.xml

The Memory Cache

Start of /roxen/2.1/programmer/concepts/ramcache.xml

If all pages were static, they would always end up in the cache. Of course, the world is seldom that simple, and the cache needs to know a little bit more about just what pages can or can not be served from a previous rendering to gain some speed, so it does not cache all too enthusiastically. It is up to you as a programmer to help it make the proper decisions.

The cache is keyed with the raw URL (including prestates and query variables), and only `GET` requests without Authentication headers are cached. Since this also includes dynamic pages with the result of database queries and the like, such pages need to interact with the cache to make sure they are not overcached. This is done in pike with the two macros `NOCACHE()` and `CACHE(`*seconds*`)`. In Java, it is done via the `cache(`*seconds*`)` and `noCache()` methods of the *RoxenRequest class*. For the Perl and CGI modules, all pages are currently never ever cached (both modules use the `NOCACHE()` macro internally).

If `NOCACHE()` has not been called sometime during the proceedings of a request, the reply is cached in the in-memory request-cache. Hence if, later on, a request for the same file with the same query and prestate parameters occurs, the in-memory entry will be used instead.

`CACHE(`*seconds*`)` limits the cache time to the number of seconds specified (`NOCACHE()` is actually just an alias for `CACHE(0)`). Hence if the same page gets called for within the number of seconds specified in the `CACHE()` macro, it will get sent from the cache. If, on the other hand, more time has passed, it will be regenerated.

This way, you may have as dynamic or as static pages as you like, and still always get optimum use of the memory cache to speed up things. Just remember that it is always up to you to instruct the cache of the cachability of results produced by your tags, modules or scripts, since it will assume that they can be put in the cache unless otherwise noted!

You can, if you want closer manual supervision of what leaves the cache, you may install a per-page callback with

```
Roxen.add_cache_callback(RequestID id,
                         function(RequestID,object:
int) callback)
```

to add a callback that is called each time the file is to be sent from the cache. If your callback returns `1`, the in-memory entry will be reused, otherwise not. If you also `destruct()` the second argument, the entry will be removed from the cache.

Another alternative is

```
Roxen.add_cache_stat_callback(RequestID id, string
file, int mtime)
```

This will case the (real) file specified as 'file' to be stat(2)ed for each request, if it's mtime differs from the supplied mtime, the entry is deleted from the cache. The 'main' file is always handled in this manner.

End of /roxen/2.1/programmer/concepts/ramcache.xml

Important Classes

Important classes; their respective use, purpose, contexts etc. Class name case kept intact in file names, as a pedagogic measure.

End of /roxen/2.1/programmer/classes/index.xml

RequestID

Start of /roxen/2.1/programmer/classes/RequestID.xml

The request information object contains all request-local information and server as the vessel for most forms of intercommunication between modules, scripts, RXML and so on. It gets passed round to almost all API callbacks worth mentioning. A RequestID object is born when an incoming request is encountered, and its life expectancy is short, as it dies again when the request has passed through all levels of the `module type calling sequence`.

Note! There are actually different request information objects for different protocols. To the programmer they try to look the same, but some slight differences might show up.

These are the member variables and methods of the object:

Protocol port_obj;
 The port object this request came from.

int time;
 Time of the request, standard unix time (seconds since the epoch; 1970).

string raw_url;
 The nonparsed, nontouched, non-* URL requested by the client. Hence, this path is unlike `not_query` and `virtfile` not relative to the server URL and must be used in conjunction with the former to generate absolute paths within the server. Be aware that this string will contain any URL variables present in the request as well as the file path.

int do_not_disconnect;
 Typically 0, meaning the channel to the client will be disconnected upon finishing the request and the RequestID object destroyed with it.

mapping(string:string) variables;
 Form variables submitted by the client browser, as found in the `form` scope in RXML. Both query (as found in the query part of the URL) and POST (submitted in the request body) variables share this scope, with query variables having priority over POST ones. In other words, the query part of the URL overrides whatever variables are sent in the request body.
 The indices and values of this mapping map to the names and values of the variable names. All data (names and values) are decoded from their possible transport encoding.

mapping(string:mixed) misc;
 This mapping contains miscellaneous non-standardized information, and is the typical location to store away your own request-local data for passing between modules et cetera. Be sure to use a key unique to your own application.

mapping(string:string) cookies;
 The indices and values map to the names and values of the cookies sent by the client for the requested page. All data (names and values) are decoded from their possible transport encoding.

mapping(string:string) request_headers;
 Indices and values map to the names and values of all HTTP headers sent with the request; all data has been transport decoded, and the header names are canonized (lowercased) on top of that. Here is where you look for the "user-agent" header, the "referer" [sic!] header and similar interesting data provided by the client.

mapping(string:mixed) client_var;
> The client scope; a mapping of various client-related variables, indices being the entity names and the values being their values respectively.

multiset(string) prestate;
> A multiset of all prestates harvested from the URL. Prestates are boolean flags, who are introduced in an extra leading path segment of the URL path put within parentheses, as in http://docs.roxen.com/(tables)/, this rendering a prestate multiset `(< "tables" >)`.
>
> Prestates are mostly useful for debugging purposes, since prestates generally lead to multiple URLs for identical documents resulting in poor usage of browser/proxy caches and the like. See config.

multiset(string) config;
> Much like prestates, the id->config multiset is typically used for boolean information of state supplied by the client. The config state, however, is hidden in a client-side cookie treated specially by roxen, namely the `RoxenConfig` cookie.

multiset(string) supports;
> All flags set by the supports system.

multiset(string) pragma;
> All pragmas (lower-cased for canonization) sent with the request. For real-world applications typically only `pragma["no-cache"]` is of any particular interest, this being sent when the user does a forced reload of the page.

string prot;
> The protocol used for the request, e g "FTP", "HTTP/1.0", "HTTP/1.1". (Se also clientprot.)

string clientprot;
> The protocol the client wanted to use in the request. This may not be the same as prot, if the client wanted to talk a higher protocol version than the server supports to date.

string method;
> The method used by the client in this request, e g "GET", "POST".

string realfile;
> When the the requested resource is an actual file in the real filesystem, this is its path.

string virtfile;
> The mountpoint of the location module that provided the requested file. Note that this is not accessable from location modules; you need to keep track of your mountpoint on your own using defvar() and query(). This mountpoint is relative to the server URL.

string rest_query;
> The scraps and leftovers of the requested URL's query part after removing all variables (that is, all key=value pairs) from it.

string raw;
> The raw, untouched request in its entirety.

string query;
> The entire raw query part (all characters after the first question mark, '?') of the requested URL.

string not_query;
> The part of the path segment of the requested URL that is below the virtual server's mountpoint. For a typical server registering a URL with no ending path component,

not_query will contain all characters from the leading '/' to, but not including, the first question mark ('?') of the URL.

string data;
> The raw request body, containing non-decoded post variables et cetera.

string remoteaddr;
> The client's IP address.

string host;
> The client's hostname, if resolved.

Stdio.File connection()
> Returns the file descriptor used for the connection to the client.

Configuration configuration()
> Returns the `Configuration` object of the virtual server that is handling the request.

End of /roxen/2.1/programmer/classes/RequestID.xml

Configuration

Start of /roxen/2.1/programmer/classes/Configuration.xml

The configuration object, reachable from the RequestID object via `RequestID->configuration()`, is roxen's internal representation of a virtual server. Within it, you find all loaded modules, methods for them to interconnect to one another and various data about the server itself.

These are the member variables and methods of the object:

Server Info

int requests;
> The number of requests, for debug and statistics info only.

int sent;
> Bytes data sent.

int hsent;
> Bytes headers sent.

int received;
> Bytes data received.

mapping(string:RoxenModule) modules;
> All enabled modules in this virtual server. The format is `"module":{ "copies":([ num:instance, ... ]) }`

mapping(RoxenModule:string) otomod;
> A mapping from the module objects to module names.

int save_one(RoxenModule o)
> Save all variables in a given module.

void save(int|void all)
> Save this configuration. If all is included, save all configuration global variables as well, otherwise only all module variables.

int(0..1) is_file(string virt_path, RequestID id)
> Is `virt_path' a file in our virtual filesystem?

int|string try_get_file(string s, RequestID id, int|void status, int|void nocache, int|void not_internal)
> Convenience function used in quite a lot of modules. Tries to read a file into memory, and then returns the resulting string.

NOTE: A 'file' can be a cgi script, which will be executed, resulting in a horrible delay.

Unless the not_internal flag is set, this tries to get an external or internal file. Here "internal" means a file that never should be sent directly as a request response. E.g. an internal redirect to a different file is still considered "external" since its contents is sent directly to the client. Internal requests are recognized by the id->misc->internal_get flag being non-zero.

string real_file(string file, RequestID id)

Return the _real_ filename of a virtual file, if any.

mapping get_file(RequestID id, int|void no_magic, int|void internal_get)

Return a result mapping for the id object at hand, mapping all modules, including the filter modules. This function is mostly a wrapper for low_get_file().

mapping|int(-1..0) low_get_file(RequestID id, int|void no_magic)

The function that actually tries to find the data requested. All modules except last and filter type modules are mapped, in order, and the first one that returns a suitable response is used. If `no_magic' is set to one, the internal magic roxen images and the find_internal() callbacks will be ignored.

The return values 0 (no such file) and -1 (the data is a directory) are only returned when `no_magic' was set to 1; otherwise a result mapping is always generated.

Inter-module Communication

array(string) userinfo(string u, RequestID|void id)

Fetches user information from the authentication module by calling its userinfo() method. Returns zero if no auth module was present.

array(string) userlist(RequestID|void id)

Fetches the full list of valid usernames from the authentication module by calling its userlist() method. Returns zero if no auth module was present.

array(string) user_from_uid(int u, RequestID|void id)

Return the user data for id u from the authentication module. The id parameter might be left out if FTP. Returns zero if no auth module was present.

mixed call_provider(string provides, string fun, mixed ... args)

Maps the function "fun" over all matching provider modules and returns the first positive response.

array(mixed) map_providers(string provides, string fun, mixed ... args)

Maps the function "fun" over all matching provider modules.

array(RoxenModule) get_providers(string provides)

Returns an array with all provider modules that provides "provides".

RoxenModule get_provider(string provides)

Returns the first provider module that provides "provides".

RoxenModule|string find_module(string name)

Return the module corresponding to the name (eg "rxmlparse", "rxmlparse#0" or "filesystem#1") or zero, if there was no such module.

End of /roxen/2.1/programmer/classes/Configuration.xml

Variable.Variable

Start of /roxen/2.1/programmer/classes/Variable.xml

The classes in the `Variable` module are the abstractions of *module variables*, with methods to define, set, read, render modification widgets for, change visibility of the variable and so on. The `Variable.Variable` class is the basic variable type in Roxen. All other variable types inherit (and should inherit) this class.

Of most interest to the roxen module programmer, we have the basic, high-level API of the `Variable` classes - how to define a variable, change its visibility check callback or get callbacks when the variable is changed:

void create(mixed default, void|int flags, void|string|object std_name, void|string|object std_doc)

The constructor, for defining and initializing variables. *Default* is the default value for the variable, *flags* is a bitwise or of one or more of the VAR_ defines defined in `module.h`:

> #### VAR_INITIAL
> Should be configured initially.
>
> #### VAR_MORE
> Only visible when more-mode is on (default on)
>
> #### VAR_DEVELOPER
> Only visible when devel-mode is on (default on)
>
> #### VAR_EXPERT
> Only for experts

The *std_name* and *std_doc* are the name and documentation string for the default locale (always english).

void set_invisibility_check_callback(function(RequestID,Variable:int) cb)

If the function passed as argument returns `1`, the variable will not be visible in the configuration interface.

Pass `0` to remove the invisibility callback.

function(RequestID,Variable:int) get_invisibility_check_callback()

Return the current invisibility check callback.

int check_visibility(RequestID id, int more_mode, int expert_mode, int devel_mode, int initial, int|void variable_in_cfif)

Return 1 if this variable should be visible in the administration interface. The default implementation checks the *flags* field, and the invisibility callback, if any. See `get_flags()`, `set_flags()` and `set_invisibility_check_callback()`.

If *variable_in_cfif* is true, the variable is in a module that is added to the configuration interface itself. This allows you to hide variables that would break something in the admin interface, if set to an value that is illegal in that context, for instance. It might seem as though you can not add modules of your own to the admin interface (and for a good reason too - tampering with the admin interface can easily lead to a malfunctioning admin environment - but if you really know what you are doing, you can start the server with the define `YES_I_KNOW_WHAT_I_AM_DOING`, and may then tamper to your heart's delight :-).

void set_flags(int flags)

Set the flags for this variable. See `create()` for more info.

int get_flags()

Returns the *flags* field for this variable. See `create()` for more info.

void set_warning(string to)

Set the warning shown in the configuration interface.

void add_warning(string to)

Like `set_warning()`, but adds to the current warning, if any.

int set(mixed to)

Sets the variable to a new value.

The return value depends on the result of the set:

true (non-zero)

The set was successful and changed the value of the variable. More precisely, the return value hints about how it was changed:

-1

The variable was changed back to its default value.

1

The variable was changed otherwise.

false (zero)

The set didn't change the value of the variable, for some reason or other:

0

A plain `0` is returned if the variable was not changed by the set.

UNDEFINED

The set failed (`verify_set()` threw a string). (`UNDEFINED` is a `0` with `zero_type` set to one, such as `([])[0]`). If `verify_set()` threw an exception, the exception is thrown, instead of returning a value.

int low_set(mixed to)

Forced set. No checking is done whatsoever. Returns:

-1

The variable was changed back to its default value.

1

The variable was changed otherwise.

0

The value was unchanged, or the set failed.

mixed query()

Returns the current value for this variable.

int is_defaulted()

Return true if this variable is set to its default value.

string get_warnings()

Returns the warnings currently signalled by the variable, if any (zero otherwise).

function(Variable:void) get_changed_callback()

Return the callback set with `set_changed_callback().`

void set_changed_callback(function(Variable:void) cb)

The function passed as an argument will be called when the variable value is changed.

Pass `0` to remove the callback.

void add_changed_callback(function(Variable:void) cb)

Add a new callback to be called when the variable is changed. If `set_changed_callback()` is called, callbacks added with this function are overridden.

mixed default_value()

Returns the default (initial) value for this variable.

When implementing your own variable types, there is a type constant and a host of other methods that are of interest too. This is also the recommended way if you are extending a variable type to provide more error checking, for instance, apart from when you use some kind of data that is not already available among the default set of variables. Just inherit a suitable base class (`Variable.Variable` if you are still doing everything from scratch), and redefine whatever you need.

string type

Mostly used for debug (`sprintf("%O", variable_obj)` uses it).

string doc()

Return the documentation for this variable (locale dependant).

The default implementation queries the locale object in roxen to get the documentation.

string name()

Return the name of this variable (locale dependant).

The default implementation queries the locale object in roxen to get the documentation.

string type_hint()

Return the type hint for this variable. Type hints are generic documentation for this variable type, and is the same for all instances of the type.

array(string|mixed) verify_set_from_form(mixed new_value)

Like `verify_set()`, but only called when the variables are set from a form.

array(string|mixed) verify_set(mixed new_value)

Return (`{ error, new_value }`) for the variable, or throw a string.

If error != 0, it should contain a warning or error message. If new_value is modified, it will be used instead of the supplied value.

If a string is thrown, it will be used as a error message from set, and the variable will not be changed.

mapping(string:string) get_form_vars(RequestID id)

Return all form variables prefixed with `path().`

mixed transform_from_form(string what)

Given a form value, return what should be set. Used by the default `set_from_form()` implementation.

void set_from_form(RequestID id)

Set this variable from the form variable in id->Variables, if any are available. The default implementation simply sets the variable to the string in the form variables.

Other side effects: Might create warnings to be shown to the user (see `get_warnings()`).

Calls `verify_set_from_form()` and `verify_set().`

string path()

A unique identifier for this variable. Should be used to prefix form variable names.

Unless this variable was created by `defvar()`, the path is set by the configuration interface the first time the variable is

to be shown in a form. This function can thus return 0. If it does, and you still have to show the form, call `set_path()` with a unique string.

void set_path(string to)
Set the path. Not normally called from user-level code.
This function must be called at least once before `render_form()` can be called (at least if more than one variable is to be shown on the same page). This is normally done by the configuration interface.

string render_form(RequestID id, void|mapping additional_args)
Return a (HTML) form to change this variable. The name of all `<input>` or similar variables should be prefixed with the value returned from the `path()` function.

string render_view(RequestID id)
Return a "view only" version of this variable.

End of /roxen/2.1/programmer/classes/Variable.xml

Available Variable Types

Start of /roxen/2.1/programmer/classes/variable_types.xml

The classes in the `Variable` module are the abstractions of *module variables*, with a common set of methods (covered in greater depth with *Variable.Variable*) and some data type dependent add-ons to constrain or otherwise customize the variable.

Variable.Flag
An on/off toggle.

Variable.Int
Integer variable, with optional range checks

> **void set_range(int minimum, int maximum)**
> Set the range of the variable, if minimum and maximum are both 0 (the default), the range check is removed.

Variable.Float
Float variable, with optional range checks, and adjustable precision.

> **void set_range(float minimum, float maximum)**
> Set the range of the variable, if minimum and maximum are both 0.0 (the default), the range check is removed.

> **void set_precision(int prec)**
> Set the number of _decimals_ shown to the user. If prec is 3, and the float is 1, 1.000 will be shown. Default is 2.

Variable.String
String variable.

> **constant width = 40;**
> The width of the input field. Used by overriding classes.

Variable.Password
Password variable (uses crypt) (extends `Variable.String`).

Variable.File
A filename (extends `Variable.String`).

string read()
Read the file as a string.

Stat stat()
Stat the file.

Variable.Location
A location in the virtual filesystem (extends `Variable.String`).

Variable.URL
A URL (extends `Variable.String`).

Variable.Directory
A Directory (extends `Variable.String`).

> **Stat stat()**
> Stat the directory.

> **array get()**
> Return a listing of all files in the directory.

Variable.Text
Text (multi-line string) variable (extends `Variable.String`).

> **constant cols = 60;**
> The width of the textarea.

> **constant rows = 10;**
> The height of the textarea.

Variable.MultipleChoice
Base class for multiple-choice (one of many) type variables.

> **void set_choice_list(array to)**
> Set the list of choices.

> **array get_choice_list()**
> Get the list of choices. Used by this class as well. You can overload this function if you want a dynamic list.

> **void set_translation_table(mapping to)**
> Set the lookup table.

> **mapping get_translation_table()**
> Get the lookup table. Used by this class as well. You can overload this function if you want a dynamic table.

> **static string _name(mixed what)**
> Get the name used as value for an element gotten from the `get_choice_list()` method.

> **static string _title(mixed what)**
> Get the title used as description (shown to the user) for an element gotten from the `get_choice_list()` method.

Variable.StringChoice
Select one of many strings (extends `Variable.MultipleChoice`).

Variable.IntChoice
Select one of many integers (extends `Variable.MultipleChoice`).

Variable.FloatChoice
Select one of many floating point (real) numbers (extends `Variable.MultipleChoice`).

void set_precision(int prec)
>> Set the number of _decimals_ shown to the user. If *prec* is 3, and the float is 1, 1.000 will be shown.
>> Default is 2.

Variable.FontChoice

Select a font from the list of available fonts (extends `Variable.MultipleChoice`).

Variable.List

The List baseclass, offering many-of-one-type types.

>> **string transform_to_form(mixed what)**
>>> Override this function to do the value->form mapping for individual elements in the array.

Variable.DirectoryList

A list of directories (subclass of `Variable.List`).

Variable.StringList

A list of strings (subclass of `Variable.List`).

Variable.IntList

A list of integers (subclass of `Variable.List`).

Variable.FloatList

A list of floating point numbers (subclass of `Variable.List`). See also `Variable.Float`.

>> **void set_precision(int prec)**
>>> Set the number of _decimals_ shown to the user. If prec is 3, and the float is 1, 1.000 will be shown.
>>> Default is 2.

Variable.URLList

A list of URLs (subclass of `Variable.List`).

Variable.PortList

A list of Port URLs (subclass of `Variable.List`).

Variable.FileList

A list of filenames (subclass of `Variable.List`).

End of /roxen/2.1/programmer/classes/variable_types.xml

The Roxen Module API

The different types of roxen modules in-depth.

End of /roxen/2.1/programmer/roxen-modules/index.xml

Introduction to Roxen Modules

Start of /roxen/2.1/programmer/roxen-modules/introduction.xml

Roxen's native programming interface is available through the roxen module API. Modules can be conceptually categorized by their type; parser modules (unformally also known as "tag modules") add custom tags to the RXML parser, location (or filesystem) modules serve entire documents in a portion of the virtual namespace of the server, logger modules take notes of the traffic passing through the server and so on. Depending on what task a module is trying to accomplish, it will be of one or more of the available module types. The module type establishes the relationship between roxen and the module, and determines when and how the module will be called into action.

All modules have access to the entire roxen API, and there is no difference between a module delivered with roxen and a module developed by a third party.

Roxen modules are `.pike` files implementing at least the methods required by its module types(s), and probably others as well. All modules must inherit `module.pike`, which besides implementing stubs for API functions marked as "optional" in this manual, also adds mnemonic names for various constants, among others, the types.

The rest of this chapter is devoted to describing the specifics of the array of module types, their API methods and where they fit in the big picture.

End of /roxen/2.1/programmer/roxen-modules/introduction.xml

The Module Type Calling Sequence

Start of /roxen/2.1/programmer/roxen-modules/calling_sequence.xml

This is an in-depth rehash of the module type calling sequence description discussed briefly earlier. As already noted, a request generally sifts down through the various type levels, until some module handles it or it ends up a file not found error.

A failure usually means that the request is passed on to the next level. What happens when a module succeeds in treating the request depends on the level, and will be described in each case.

1. First off, the request in intercepted on the socket by a protocol module. The protocol module (not located in `server/modules/` as the other modules shipped with roxen; the protocol modules are found in `server/protocols/` and are spawned off from the stub classes in `server/roxen.pike`) handles the low-level talking to the client in the protocol at hand, resolves what virtual server should handle the request, and gets the wheel rolling. The protocol module object instance is the RequestID object which will tag along for the rest of the request, bearing all request-local state.

2. Before leaving the protocol module, sending the request forth to the various module types, the RAM cache is checked for an already rendered version of the page. If the cache had an equivalent entry that didn't need to be regenerated, it gets sent right away, giving us a nice speed boost, in comparison with the time it would take to render it all over again. What constitutes "an equivalent entry" is covered in greater detail where *The Memory Cache* is explained. A cached request then gets promoted to the final (logging) stage in this list, the rest sift down through the rest of the module types.

3. Next stop in the chain is the authentication module, when present. The authentication module has a look on the request object, sets some flags and regardless of the success or failure, the request moves on. It is noteworthy that this level is only entered when the client sent an authorization header, so authentication modules based on other criteria probably also double as some other module type(s).

4. The next class of modules that see the request are the First modules. If a first try module would decide to handle the request on its own, the request is passed along to the filter level further down the chain.

5. Still unhandled requests now undergo a check for what path was accessed, and is then routed to location modules accordingly, until a response is generated or there are no location modules left. After this, a non-handled request move on to the filter level. A handled request returning a Stdio.File object is forwarded to the extension module level and a returned directory indicator is passed on to the directory listing module. All other results are forwarded to the filter level.

6. The request was considered a directory, and it is up to the directory listing module to generate some form of directory listing or representation of the directory at hand. Regardless of success or failure to do so, the request moves on to the filter level.

7. Stdio.File objects generated by previous levels are sent to the file extension modules according to the filename extension of the virtual file. File Extension modules are typically used to implement interfacing to external processes, such as a CGI engine. Regardless of success, the request will be passed on to the filter level.

8. The content-type module, although mostly orthogonal to the request path calling chain, possibly being hailed from any module, also may be invoked automatically by roxen. This happens in the event of a Stdio.File object being returned from the extension module level. The content-type module is then invoked to devise a content-type for the result based on the name of the virtual file accessed.

9. All requests then pass through the filter module stage for possible processing of the returned data before sending away the result to the browser. It is worth noticing that the state of the request at this stage could differ a lot; for instance a directory listing, an open file descriptor or even the yet unhandled request (candidate to become a file not found message). If the request is still not handled after the filter module stage, it is forwarded to the MODULE_LAST type stage, otherwise it is forwarded to the protocol module once more.

10. If the entire module type calling sequence passed so far has failed to return a response, the last modules will have a final shot at catching the request before roxen returns a file not found error. Regardless of result, the request is forwarded from the MODULE_LAST stage to the protocol module.

11. The protocol module which originally set this chain going is returned the result from previous stages and starts sending the result to the client in response to the request.

12. Once the protocol module has initiated the response transfer to the client, the logger modules get a shot at logging some information in some way, until one of them decides that the request shall not see any other log modules or there are no more loggers left. Since the request has already been sent (or is just being sent), the logger modules can not alter the response seen by the client.

End of /roxen/2.1/programmer/roxen-modules/ calling_sequence.xml

Constants Common to All Modules

Start of /roxen/2.1/programmer/roxen-modules/ common_constants.xml

This page covers all constants that all module types may or are expected to provide with information about the module. (Be sure to see the sections on common *callback methods* and *API methods* too.)

string cvs_version;
   This string (filtered to remove some ugly cvs id markup) shows up in the roxen administration interface when handling module parameters in developer mode (configured under "User Settings" below the Admin tab). It will also serve as the basis for extracting version information of the file in the inherit tree. Optional, but convenient, especially if you use cvs for version control of your code.

string module_name;
   The name that will show up in the module listings when adding modules or viewing the modules of a virtual server. Keep it fairly informative and unique, since this is the only means for identification of your module in the most brief add module view mode.

int module_type;
   Module type (see server/etc/include/module.h). May be bit-wise ored (|) for hybrid modules. Hybrid modules must implement the required API functions for all of the module types they are hybrids of.

string module_doc;
   The documentation string that will end up in the administration interface view of the module just below the module name. Also shows up in the more verbose add module views.

int module_unique;
   0 to allow multiple instances of the module per virtual server, 1 to allow at most one.

int thread_safe;
   Tell Roxen that this module is thread safe. That is, there is no request specific data in module global variables (such state is better put in the *RequestID* object, preferably in the id->misc mapping under some key unique to your module).
   If your module is not thread safe, setting this flag to zero (0) or leaving it unset altogether will make roxen serialize accesses to your module. This will hurt performance on a busy server. A value of one (1) means your module is thread safe.

End of /roxen/2.1/programmer/roxen-modules/ common_constants.xml

Callback Methods Common to All Modules

Start of /roxen/2.1/programmer/roxen-modules/ common_callbacks.xml

This page covers all callbacks that all module types may or are expected to implement in order to function as a proper roxen module. (Be sure to see the sections on common *constants* and *API methods* too.)

void create(Configuration|void conf)
   In create() , you typically define your module's configurable variables (using the *defvar()* method) and set data about it using set_module_creator() and set_module_url(). The *configuration object* of the virtual server the module was initialized in is always passed, except for the one occasion when the file is compiled for the first time, when the `conf' argument passed is 0. Se also start().

string info(Configuration|void conf)
   Implementing this function in your module is optional.
   When present, it returns a string that describes the module. When absent, Roxen will use element module_doc. Unlike module_doc, though, this information is only shown when the module is present in a virtual server, so it won't show up when adding modules to a server.

void start(int occasion, Configuration conf)
   Set up shop / perform some action when saving variables. (optional)
   If occasion is 0, we're being called when starting up the module, to perform whatever actions necessary before we are able to service requests. This call is received when the virtual server the module belongs to gets initialized, just after the module is successfully added by the administrator or when reloading the module.
   This method is also called with occasion set to 2 whenever the configuration is saved, as in when some module variable has changed and the administrator clicked "save" in the admin interface. This also happens just before calling stop() upon reloading the module.

void stop()
   Close down shop. (optional)

Tidy up before the module is terminated. This method is called when the administrator drops the module, reloads the module or when the server is being shut down.

string status()

Tells some run-time status, statistics or similar to the curious administrator. Implementing this function is optional.

Returns a string of HTML that will be put in the admin interface on the module's settings page just below the `documentation string`.

mapping|int(-1..0)|Stdio.File   find_internal(string   file, RequestID id)

Internal magic location any module may use, similar to the `find_file()` method of all *Location (Filesystem) Modules*, but with a less civilized-looking URL (typically prefixed with the string `"/internal/"` prepended to your module identifier). This method is mostly used for background things where a URL doesn't show too much, such as when generating background images and the like. To get the internal mountpoint below which find_internal will handle requests, use `query_internal_location()`.

May return
- (a normal response mapping)
- (the integer value `-1`.)
- (an Stdio.File object)
- signalled by the integer value `0`.

End   of   /roxen/2.1/programmer/roxen-modules/ common_callbacks.xml

API Methods Common to All Modules

Start   of   /roxen/2.1/programmer/roxen-modules/ common_api.xml

This page covers all API methods available to all (and interesting to all) module types. (Be sure to see the sections on common *constants* and *callbacks* too.)

int module_dependencies(Configuration conf, array(string) modids)

Add these modules to the configuration; a handy way of assuring that modules whose presence your module depends on are really there. *modids* is an array of module identifiers as those used by `Configuration->find_module()`. Typical usage:

mapping(string:function(RequestID:void)) query_action_buttons(RequestID id)

Optional callback for adding action buttons to the module's administration settings page; convenient for triggering module actions like flushing caches and the like.

The indices of the returned mapping are the action descriptions that will show up on each button (e g "flush cache"), and the corresponding values are the callbacks for each button respectively. These functions may take an optional RequestID argument, where this id object refers to the id object in the admin interface sent with the request used to invoke the action by the administrator.

string query_internal_location()

Returns the internal mountpoint, where `find_internal()` is mounted.

string query_absolute_internal_location(RequestID id)

Returns the internal mountpoint as an absolute path. This includes the site prefix, which is typically the empty string

for a site mounted at a URL with no path component, such as `"http://*/"`, as opposed to for example `"http://*/ roxen/2.1/"`, which would be prefixed "roxen/2.1/".

This method is recommended when you want to generate a link to your internal resources supplied by `find_internal()`.

void set_module_creator(string|array(string) c)

Set the name and optionally email address of the author of the module. Names on the format `"author name <author_email>"` will end up as links on the module's information page in the admin interface. In the case of multiple authors, an array of such strings can be passed to generate one link per author (the array order is preserved).

void set_module_url(string to)

A common way of referring to a location where you maintain information about your module, or similar. The URL will turn up on the module's information page in the admin interface, referred to as the module's home page.

End   of   /roxen/2.1/programmer/roxen-modules/ common_api.xml

Module Variables

Start of /roxen/2.1/programmer/roxen-modules/defvar.xml

Roxen modules have other common denominators than those mentioned on the previous pages - the concept of administrator configurable variables for whatever module configuration data your module might depend on. Besides the variable types mentioned here, you may extend Roxen with user-defined data types, for which you create the widgetry, define storage methods and so on.

The module variables accessible from the administration interface are defined using `defvar()`, queried using `query()` or `QUERY()`, can be undefined using `killvar()`, may be altered run-time using `set()` or tweaked with using the variable object's own methods, after getting hold of the variable object with `getvar()`.

Each roxen module may or may not choose to define any number of variables, each having a name unique to the module (to identify it when referring to it, polling or modifying its value, visibility or similar). The variable definition is shared among all instances of your module, but each module may have been set up with different operating parameters by the administrator, so their values may differ. The configuration data resides with the configuration data of the virtual server that the module is loaded in, where each instance of a module has its own region.

As a technical curiosity, it can be noted that module variables are quite memory conservative - even a huge roxen server setup with thousands of virtual servers, each with several modules loaded, won't gnaw away as much memory as would an older roxen (or any caudium) server. This is possible, since module variables that have not changed from their preset values (which is typically the case, since most variables have reasonable defaults), may fall back on that value, which can be found in the module's definition, instead of among its data.

In short: if you encounter some situation where there is reason to provide an administrator a choice of operating parameters, don't make constant declarations in your code - define module variables instead. It's easy, both for you as a programmer and for the administrator, more fault-tolerant (not having anybody meddling with your code), as robust as you care to

make it (you may equip your variable objects with hooks to verify input validity, should the datatype be too lax for your own taste and purposes), it provides an easy way to document your variables, only show those variables that are relevant given the circumstances and finally - it even looks nice! (Never underestimate the value of readable code. :-)

Variable.Variable defvar(string name, Variable.Variable var)

Defines a module variable *var*, giving it the unique identifier *name* (this name is used when accessing the variable with `query()`, `set()` and friends). The recommended place to put your `defvar()` declarations is in your module's `create()` callback, such as:

For info on the available variable types, and the full host of API methods available in the variable objects themselves, see the docs for *Variable.Variable*.

In roxen versions 2.0 and older, the `defvar()` API looked a bit differently (and the old-style API is still legal, for backwards compatibility) - it is mentioned here only for ease of reading and understanding legacy code:

Variable.Variable defvar(string var, mixed value, string name, int type, string doc, array|void misc, int|function|void hide_if_true)

A deprecated (although it will be supported for the forseeable future) compatibility version of the function. This is how you read its parameters:

string var
The variable identifier

mixed value
The default value

string name
The name of the variable that gets presented in the admin interface

int type
The type of the variable (one of the TYPE_ defines defined in `module.h`). These constants correspond to the various `Variable.Variable` classes (similarly named minus the TYPE_ prefix) used in the new API.

string doc
The variable documentation string

array|void misc
Additional type-dependent data

int|function|void hide_if_true

mixed query(string var, int|void ok)
Query the value of the variable *var*. If *ok* is true, it is not an error if the specified variable does not exist. Instead, UNDEFINED (the value `0` with zero_type()==1) is returned.

QUERY(var)
Compatibility/convenience macro defined in `module.h` that is synonym to `query()`, apart from its syntax. If you prefer the syntax QUERY(var) to query("var"), feel free to use that instead. There used to be a speed difference in older versions of roxen, where the QUERY() method was slightly faster (didn't perform any error checking), but that is no longer the case.

int killvar(string var)
Undefine the previously defined variable *var*. This is useful mainly when inheriting another module (`filesystem.pike`,

for instance), and all of its variables do no apply to your extended or otherwise modified version.

void set(string var, mixed value)
Set the variable *var* to the specified value. Typically used when you have invisible variables where you save some form of module state that should survive a server restart, but that is not meaned to be modified by an administrator.

Variable.Variable getvar(string name)
Return the variable object associated with the specified variable.

End of /roxen/2.1/programmer/roxen-modules/defvar.xml

Tag Modules

Start of /roxen/2.1/programmer/roxen-modules/tag.xml

The module type constant for tag modules is MODULE_TAG.

Extending RXML with your own tags is done by implementing tag modules. There are two methods of writing tags; simpletags being by far the easier one, RXML.pmod class style being the more powerful of the two.

Writing simpletags is mostly a matter of naming your module's methods; methods with a leading "simpletag_" will be caught as tag definitions, and the tag name constructed from the function name (after underscores having been changed into dashes).

A tag definition such as

```
string simpletag_my_test_tag(string name, mapping a
rg, string contents, RequestID id)
```

will generate a tag `<my-test-tag>`, and when encountered in an RXML parsed document (for example, `<my-test-tag testing='yes'>hi.</my-test-tag>`), the function will be called with the arguments filled with, in turn, the name of the tag (`"my-test-tag"`), a mapping of its arguments (`([ "testing":"yes" ])`), its contents (`"hi."`) and the RequestID object. The string returned by the function will be inserted in the document, replacing the tag and its contents.

By providing an integer

```
int simpletag_my_test_tag_flags = RXML.FLAG_DEBUG
```

you can alter how the RXML parser calls and treats your tag, by oring together the flags of your choice from the list below.

RXML Tag Flags
- Write a lot of debug during the execution of the tag, showing what type conversions are done, what callbacks are being called etc. Note that DEBUG must be defined for the debug printouts to be compiled in (normally enabled with the --debug flag to Roxen).
- Flags this as a processing instruction tag (i.e. one parsed with the <?name ... ?> syntax in XML). The string after the tag name to the ending separator constitutes the content of the tag. Arguments are not used.

  A still easier way of making processing instruction tags is by calling your tag function simple_pi_tag_*my_tag_name* right away. the argument template for the tag is the same, but the argument mapping sent is always empty, for obvious reasons.
- If set, the tag does not use any content. E.g. with an HTML parser this defines whether the tag is a container or not, and in XML parsing it simply causes the content (if any) to be thrown away.

- Makes the PXml parser parse the tag in an HTML compatible way: If FLAG_EMPTY_ELEMENT is set and the tag doesn't end with '/>', it will be parsed as an empty element. The effect of this flag in other parsers is currently undefined.
- Never apply any prefix to this tag.
- Don't preparse the content with the PXml parser. This is always the case for PI tags, so this flag doesn't have any effect for those. This is only used in the simple tag wrapper. Defined here as placeholder.
- Postparse the result with the PXml parser. This is only used in the simple tag wrapper. Defined here as placeholder.

Class-based Tag API

To be continued...

End of /roxen/2.1/programmer/roxen-modules/tag.xml

Location (Filesystem) Modules

Start of /roxen/2.1/programmer/roxen-modules/location.xml

```
mapping|Stdio.File|int(-
1..0) find_file( string path, RequestID id )
```

The return value is either a response mapping, an open file descriptor or 0, signifying that your module did not handle the request. Return -1 to indicate the resource being a directory.

This is the fundamental method of all location modules and is, as such, required. It will be called to handle all accesses below the module's mount point. The path argument contains the path to the resource, in the module's own name space, and the id argument contains the request information object.

That the path is in the modules name space means that the path will only contain the part of the URL after the module's mount point. If a module is mounted on `/here/` and a user requests `/here/it/is.jpg`, the module will be called with a path of `it/is.jpg`. That way, the administrator can set the mount point to anything she wants, and the module will keep working. Note that changing the mount point to `/here` would give the module the path `/it/is.jpg` for that request.

A zero return value means that the module could not find the requested resource. In that case roxen will move on and try to find the resource in in other location modules. Returning -1 means that the requested resource is a directory, in which case the request will be handled by a directory type module.

If the module could handle the request, the return value is either a response mapping or a Stdio.File object containing the requested file.

```
string query_location()
```

Returns the location in the virtual server's where your location module is mounted. If you make a location module and leave out this method, the default behaviour inherited from module.pike will return the value of the module variable 'location'.

```
Stat stat_file( string path, RequestID id )
```

The stat_file() method emulates Pike's `file_stat()` method, returning information about a file or directory. path is the path to the file or directory in the module's name space, id is the request information object.

stat_file() is most commonly used by directory type modules to provide informative directory listings, or by the ftp protocol module to create directory listings.

The return value it is expected to be a Stat array (or, in future versions, a Stdio.Stat object, for instance created from such an array):

```
({ mode, size, atime, mtime, ctime, uid, gid })
```

`mode` is an integer containing the unix file permissions of the file. It can be ignored.

`size` is an integer containing the size of the file, or a special value in case the object is not actually a file. Minus two means that it is a directory, minus three that it is a symbolic link and minus four that it is a device special file. This value must be given.

`atime` is a unixtime integer containing the last time the file was accessed (seconds since 1970). It can be ignored.

`mtime` is a unixtime integer containing the last time the file was modified (seconds since 1970). It will be used to handle Last-Modified-Since requests and should be supplied if possible.

`ctime` is a unixtime integer containing the time the file was created (seconds since 1970). It can be ignored.

`uid` is an integer containing the user id of this file. It will be correlated with the information from the current authentification type module, and used by the CGI executable support module to start CGI scripts as the correct user. It is only necessary for location modules that provide access to a real file system and that implement the real_file() method.

`gid` is an integer containing the group id of the file. It is needed when uid is needed.

```
mapping(string:Stat) find_dir_stat( string path, Re
questID id )
```

Need not be implemented. The parameter `path' is the path to a directory, `id' is the request information object and the returned mapping contains all filenames in the directory mapped to Stat objects for the same files respectively.

If this method is not implemented, the find_dir_stat function inherited from `module.pike` maps the result of find_dir() over stat_file() to produce the same result. Providing your own find_dir_stat might be useful if your module maps its files from a database, in which case you would gain performance by using just one big query instead of hordes of single-file queries.

```
string|void real_file( string path, RequestID id )
```

This method translates the path of a file in the module's name space to the path to the file in the real file system. path is the path to the file in the module's name space, id is the request information object.

If the file could not be found, or the file doesn't exist on a real file system, zero should be returned. Only location modules that access server files from a real file system need implement this method. See also the stat_file() method.

```
array(string)|void find_dir( string path, RequestID
 id )
```

The find_dir() returns a directory listing; an array of strings containing the names of all files and directories in this directory. The path parameter is the path to the directory, in the module's name space, id is the request information object.

This method is usually called because a previous call to find_file() returned that this path contained a directory and a directory type module is right now trying to create a directory listing of this directory. Note that it is possible that the find_dir() is called in several location modules, and that the actual directory listing shown to the user will be the concatenated result of all those calls.

To find information about each entry in the returned array the stat_file() is used.

End of /roxen/2.1/programmer/roxen-modules/location.xml

File Extension Modules

Start of /roxen/2.1/programmer/roxen-modules/ file_extension.xml

File extension modules handle one or several different file types. A file extension module is called after a location, or other module type, has returned a Stdio.File object with the correct extension.

The module type constant is MODULE_FILE_EXTENSION.

```
array(string) query_file_extensions()
```

Returns an array of strings containing the extensions this module handles. It should be configurable by the user, the easiest way would be to use a configuration variable of TYPE_STRING_LIST.

```
mapping handle_file_extensions( Stdio.File file, st
ring ext, RequestID id )
```

The method that will be called to do the actual work. file is the file object that a previous module returned. ext is the extension of the request, id the request information object. The return value is a response mapping.

End of /roxen/2.1/programmer/roxen-modules/ file_extension.xml

Filter Modules

Start of /roxen/2.1/programmer/roxen-modules/filter.xml

Type MODULE_FILTER modules.

```
mapping|zero filter(mapping|zero result, RequestID
id)
```

The `filter()` method is called for all requests just before the final resulting page is sent back to the browser, except when:
- The reply for a request is found in the memory cache.
- A module calls id->handle_reply() directly after accepting responsibility for the connection.

In effect, filter modules are essentially MODULE_LAST modules that get called for all requests, not only failed requests. The result parameter is either a zero (for an unhandled request) or a standard response mapping, as returned by any previous modules in the server. The id argument, as usual, is the request information object associated with the request.

The returned value is either zero, here signifying that you didn't rewrite or in any way alter the result mapping, or a new or changed result mapping.

Since all data server by your virtual server gets passed your filter module(s), you typically need to make sure your filter module doesn't interfere with such requests it wasn't intended to touch, or you may end up with some pretty hard to find problems.

End of /roxen/2.1/programmer/roxen-modules/filter.xml

Authentication Modules

Start of /roxen/2.1/programmer/roxen-modules/authentica-tion.xml

Leading note: The MODULE_AUTH API described on this page is likely subject to change in the not too distant future.

```
array|int auth(array(string) auth, RequestID id)
```

The auth method of your MODULE_AUTH type module is called when the browser sent either of the `Authorization` or `Proxy-Authorization` HTTP headers (see RFC 2617).

The auth argument passed is calculated as header_content/" ", but where the second element is base64-decoded (meaning that you won't need to do so yourself). A typical auth array you might receive could look like ({ "Basic", "Aladdin:open sesame" }), where Aladdin would be the user name the client logged in with, and "open sesame" his password.

The three elements in the returned array are, in order:

1. an int(0..1) signifying authentication failure (0) or success (1)

2. a string with the username (authenticated or not)

3. when failed, a string with the password used for the failed authentication attempt, otherwise the integer zero.

See also Roxen.http_auth_required() and Roxen.http_proxy_auth_required().

```
string user_from_uid(int uid, RequestID|void id)
```

Return the login name of the user with uid `uid'.

```
array(string) userlist(RequestID|void id)
```

Return an array of all valid user names.

```
array(string|int) userinfo(string user, RequestID|v
oid id)
```

Return /etc/passwd-style user information for the user whose login name is `user'. The returned array consists of:

```
({ login name,
   crypted password,
   used id,
   group id,
   name,
   homedirectory,
   login shell
})
```

All entries should be strings, except uid and gid, who should be integers.

End of /roxen/2.1/programmer/roxen-modules/authentica-tion.xml

Logger Modules

Start of /roxen/2.1/programmer/roxen-modules/logger.xml

A log module handles logging of requests. It can be used to log requests by other means then log files, or to disable the builtin logging for some requests.

The module type constant is MODULE_LOGGER.

```
int(0..1) log( RequestID id, mapping response )
```

id is the request information object, response is a mapping containing the response information that are about to be sent to the

browser. If the log() method returns one the logging will stop, and no other log modules will be called nor will the internal logging take place.

End of /roxen/2.1/programmer/roxen-modules/logger.xml

First Modules

Start of /roxen/2.1/programmer/roxen-modules/first.xml

A first module is called right after the authentification module. It has the opportunity of handling the whole request before the normal processing.

The module type constant is MODULE_FIRST.

```
mapping first_try( RequestID id)
```

id is the request information object. The return value is either a response mapping or zero (0) for non-handled requests.

End of /roxen/2.1/programmer/roxen-modules/first.xml

Last Modules

Start of /roxen/2.1/programmer/roxen-modules/last.xml

The module type constant for last resort modules is MODULE_LAST.

```
mapping|int(0..1) last_resort(RequestID id)
```

The `last_resort()` method is called when all previous modules have failed to return a response.

The id argument is the request information object associated with the request.

The returned value is either zero, if you didn't handle the request, a response mapping or the integer one, signifying that the request should be processed again from start (used only by the Path info support module).

End of /roxen/2.1/programmer/roxen-modules/last.xml

Provider Modules

Start of /roxen/2.1/programmer/roxen-modules/provider.xml

Provider modules are modules that provide services to other modules. The module type constant is MODULE_PROVIDER.

```
string|array(string) query_provides()
```

returns the name of the service or services this module provides, either as a string or as an array of strings.

Methods available to other modules are:

```
RoxenModule conf->get_provider( string service )
```

Returns the provider module that handles the service 'service', or one with highest priority if there are several. conf is the configuration object for the virtual server (id->configuration() fetches the current request's configuration). Any public function (or data element) can be reached via the returned module object.

```
array(RoxenModule) conf-
>get_providers( string service )
```

Returns all provider modules that handle the service service. conf is the configuration object for the virtual server.

```
void map_providers( string service, string fun, mix
ed ... args )
```

Calls the method named fun in all modules providing the service service. The method will be called with args as arguments.

```
mixed call_provider( string service, string fun, mi
xed ... args )
```

Calls the method named fun in modules providing the service service with the arguments args. Modules will get called until one module returns a non-zero value. That return value, or zero if all modules returned zero, will be returned.

End of /roxen/2.1/programmer/roxen-modules/provider.xml

Content Type Modules

Start of /roxen/2.1/programmer/roxen-modules/types.xml

The module type constant for content-type modules is MODULE_TYPES.

```
array(string|int) type_from_extension(string ext)
```

Return an array (`{ content_type, content_encoding }`) devised from the file extension `ext'. When `ext' equals "default", roxen wants to know a default type/encoding. If the content-type returned is the string `"strip"`, the content-encoding returned will be kept, and another call be made for the last-but-one file extension to get the content type (eg for `".tar.gz"` to resolve correctly).

End of /roxen/2.1/programmer/roxen-modules/types.xml

Directory Listing Modules

Start of /roxen/2.1/programmer/roxen-modules/directories.xml

A directory type module handle accesses to directories. This is usually done by creating a directory listing of the contents in the directory, or finding a suitable index file to be returned instead. There can only be one directory module in a virtual server.

The module type constant for directory modules are MODULE_DIRECTORIES.

```
mapping parse_directory( RequestID id )
```

Returns a normal response containing either a suitable directory listing or an index file. The path to the directory being listed is found in id->>not_query.

End of /roxen/2.1/programmer/roxen-modules/directories.xml

Roxen-specific Pike Modules

Roxen

Start of /roxen/2.1/programmer/pike-modules/Roxen.xml

These methods are all accessable from everywhere within roxen via Roxen.*methodname*(args). (Although technically, they don't really all belong to Roxen.pmod.)

Response Methods

Convenience functions to use in Roxen modules. When you just want to return a string of data, with an optional type, this is the easiest way to do it if you don't want to worry about the internal roxen structures.

mapping http_string_answer(string text, string|void type)
Generates a response mapping with the given text as the request body with a content type of *type* (or "text/html" if none was given).

mapping http_file_answer(Stdio.File text, string|void type, int|void len)
Generates a response mapping with request body contents drawn from the given file object with a content type of *type* (or "text/html" if none was given). If no length is supplied, it is calculated for you automatically.

mapping http_rxml_answer(string rxml, RequestID id, void|Stdio.File file, void|string type)
Parse the supplied rxml and generate a response mapping with a content type of *type* (or "text/html" if none was given).

mapping http_low_answer(int errno, string data)
Return a response mapping with the error and data specified. The error is infact the status response, so '200' is HTTP Document follows, and 500 Internal Server error, etc.

mapping http_pipe_in_progress()

mapping http_try_again(float delay)
Causes the request to be retried in delay seconds.

mapping http_redirect(string url, RequestID|void id)
Simply returns a http-redirect message to the specified URL. If the url parameter is just a virtual (possibly relative) path, the current id object must be supplied to resolve the destination URL.

mapping http_stream(Stdio.File from)
Returns a response mapping where the data returned to the client will be streamed raw from the given Stdio.File object, instead of being packaged by roxen. In other words, it's entirely up to you to make sure what you send is HTTP data.

mapping http_auth_required(string realm, string|void message)
Generates a response mapping that will instruct the web browser that the user needs to authorize himself before being allowed access. *realm* is the name of the realm on the server, which will typically end up in the browser's prompt for a name and password (e g "Enter username for *realm* at *hostname*:"). The optional message is the message body that the client typically shows the user, should he decide not to authenticate himself, but rather refraim from trying to authenticate himself.

In HTTP terms, this sends a `401 Auth Required` response with the header `WWW-Authenticate: basic realm="`*realm*`"`. For more info, see RFC 2617.

mapping http_proxy_auth_required(string realm, void|string message)
Generates a response mapping that will instruct the client end that it needs to authenticate itself before being allowed access. *realm* is the name of the realm on the server, which will typically end up in the browser's prompt for a name and password (e g "Enter username for *realm* at *hostname*:"). The optional message is the message body that the client typically shows the user, should he decide not to authenticate himself, but rather refraim from trying to authenticate himself.

In HTTP terms, this sends a `407 Proxy authentication failed` response with the header `Proxy-Authenticate: basic realm="`*realm*`"`. For more info, see RFC 2617.

Utility Functions

void set_cookie(RequestID id, string name, string value, int|void expire_time_delta, string|void domain, string|void path)
Set the cookie specified by 'name' to 'value'. Sends a Set-Cookie header.

The expire_time_delta, domain and path arguments are optional.

If the expire_time_delta variable is -1, the cookie is set to

expire five years in the future. If it is 0 or ommited, no expire information is sent to the client. This usualy results in the cookie being kept until the browser is exited.

string http_date(int t)
Returns a http_date, as specified by the HTTP-protocol standard. This is used for logging as well as the Last-Modified and Time heads in the reply.

string http_encode_string(string f)
Encode a string for inclusion in HTTP headers.

string http_encode_cookie(string f)
Encode a string for inclusion in a cookie.

string http_encode_url(string f)
Encode a string for inclusion in a URL.

void remove_cookie(RequestID id, string name, string value, string|void domain, string|void path)
Remove the cookie specified by 'name'. Sends a Set-Cookie header with an expire time of 00:00 1/1 1970. The domain and path arguments are optional.

End of /roxen/2.1/programmer/pike-modules/Roxen.xml

Pike

Pike, the native programming language of roxen, can be used in various contexts within roxen. The scope of this chapter is to explain how pike within roxen differs from stand-alone pike as well as where and how you can deploy pike code within your applications.

End of /roxen/2.1/programmer/pike/index.xml

Script

Start of /roxen/2.1/programmer/pike/script.xml

How to make pike scripts and how to use them within (and possibly also independent of) roxen.

Pike scripts differ from e g CGI or Perl scripts, by being run inside of Roxen WebServer instead of as an external process. They share the characteristics of being executed when a user tries to access them, but they have the added benefit of easy access to all Roxen API:s, being written in the server's native language. This makes them much more efficient, generally respond faster and use less resources. It is also possible for them to cache data between requests, since they will stay resident after being loaded and compiled.

Since Pike scripts are run internally in the web server they have security implications, a Pike script can do anything the web server can. It is however possible to run them in a mode where a separate process is created for each request. This is safe, but on the other hand you miss much of the advantages of Pike scripts mentioned above.

API Methods
- `create`, although not strictly a part of the Roxen API, is called once when instantiating the script during compilation, and never again.
- The method `parse` gets executed for each request to the script. It returns either a string containing RXML code or a response mapping created via one of the response methods (Roxen.http_string_answer and friends).
- Normally, scripts are kept resident within the server between request after having once been compiled. They are, as one might expect, automatically reloaded if they (or any file they have inherited) have changed on disk. This behaviour can be altered by adding a `no_reload` function to your script. The no_reload function, when present, overrules this behaviour. If it returns zero, the script is reloaded, if it returns one, it is not. It gets called before each new request after the first one (when it was originally compiled).

By default, pike scripts can be reloaded by requesting them with the pragma no-cache header. This is achieved in Netscape by pressing reload and, unfortunately, not at all, in Internet Explorer. You may easily devise any reloading policy by testing data from the id object; this example reloads the script when the query variable "reload" is set:

An example script:

```
int count = 0;
```

```
mapping parse(RequestID id)
{
  string times, reply = #"<html><head>
<title>Hello World</title>
</head><body>
<h1>Hello, world!</h1>

<p>(Did you know that I've hailed the
world %s since I was loaded?)</p>
</body></html>";

  switch(++count)
  {
    case 1: times = "just once"; break;
    case 2: times = "twice already"; break;
    default: times = "a smashing " + count + " time
s";
  }
  return Roxen.http_string_answer(sprintf(reply, ti
mes));
}
```

End of /roxen/2.1/programmer/pike/script.xml

Processing Instruction

Start of /roxen/2.1/programmer/pike/tag.xml

How to use the `<?pike ... ?>` processing instruction.

The `<?pike ... ?>` processing instruction tag (PI for short) is used to smoothly inline pike code in your pages. The perhaps most usable aspect of this is for quickly throwing together a small webpage application or for experimenting and toying around with the roxen programming environment. Since the code resides in the page, it won't be so reusable a component, as would a roxen module, but the short cycle time between applying a change and seeing the results right after another page reload make it an ideal development tool.

Methods

Three additional methods set the pike PI tag apart from those present everywhere else within roxen. These handle the output facilities of the tag:
- Outputs the string to the page. Takes the same arguments as write or sprintf in common pike, but appends the result to the output buffer. If given only one string argument, it's written directly to the output buffer without being interpreted as a format specifier.
- Returns the contents of the output buffer and resets it. This buffer contains everything written so far via write (or, by inference, the available convenience syntaxes for write) since the start of the processing instruction or the last call to flush().
- Parses the string with the RXML parser, returning the result after parsing.

Special Variables
- The RequestID object, bearing all request local state, is available in the `id' variable.

- All scopes (including _) are accessible directly as variables named after the scope itself; hence _, roxen, form, variables and so on are present for easy reading (and/or tampering :-). Thus roxen.uptime, for instance, is an integer marking the number of seconds since roxen was started, just as roxen.uptime would be in a section of RXML code (these shorthands for RXML.user_get_var and RXML.user_set_var only work for entities in scopes available in the context when the parser encountered the processing instruction).

For meshing pike, RXML code and common HTML, there are other convenience features present, in the form of certain comments treated specially. //O (the letter O) comments are written out directly into the page, and //X comments are parsed as RXML on the spot and written out into the result page. These shorthands are equivalent to the calls write(my_string) and write(rxml(my_string)) respectively.

An example pike processing instruction that shows some server info and toggles the value of a cookie:

```
<?pike
  //X <gtext>Server Info</gtext><br />
  write("This is %s running %s, and we've been up f
or %d seconds.",
                          roxen.version, roxen["pike-
version"], roxen.uptime);

  //X <br /><gtext>Cookies</gtext><br />
  //
X <pre><insert scope='cookie' variables='full' /></
pre>
  if(cookie.hi == "Hi!")
    cookie.hi = "Ho!";
  else
    cookie.hi = "Hi!";
?>
```

End of /roxen/2.1/programmer/pike/tag.xml

Java

AbstractLocationModule

Start of /roxen/2.1/programmer/java/AbstractLocation-Module.html

```
java.lang.Object
   |
  +--com.roxen.roxen.Module
        |
        +--com.roxen.roxen.AbstractLocationModule
```

An abstract adaptor class that provides default implementations for most methods in the LocationModule interface.

A module inheriting this class must either create a module variable location using defvar, or provide a different implementation of the queryLocation method.

public **AbstractLocationModule**`()`

public java.lang.String **queryLocation**`()`

public java.lang.String[] **findDir**`(java.lang.String f`
`,`
`                           RoxenRequest id)`

public java.lang.String **realFile**`(java.lang.String f,`
`                           RoxenRequest id)`

public int[] **statFile**`(java.lang.String f,`
`                     RoxenRequest id)`

End of /roxen/2.1/programmer/java/AbstractLocationModule.html

ExperimentalModule

Start of /roxen/2.1/programmer/java/ExperimentalModule.html

The interface for modules that should be marked as experimental in the module listing. It contains no methods, implementing it just flags the module as experimental.

FileExtensionModule

Start of /roxen/2.1/programmer/java/FileExtensionModule.html

The interface for modules which handle a specific file extension.

public java.lang.String[] **queryFileExtensions**`()`

public RoxenResponse **handleFileExtension**`(java.io.Fil`
`e file,`
`                          java.lang.`
`String ext,`
`                          RoxenReque`
`st id)`

End of /roxen/2.1/programmer/java/FileExtensionModule.html

Frame

Start of /roxen/2.1/programmer/java/Frame.html

```
java.lang.Object
   |
  +--com.roxen.roxen.Frame
```

An object representing an RXML parse frame

End of /roxen/2.1/programmer/java/Frame.html

HTTP

Start of /roxen/2.1/programmer/java/HTTP.html

```
java.lang.Object
   |
  +--com.roxen.roxen.HTTP
```

A support class providing HTTP related functionality. Rather than using this class directly, all these functions can be accessed through the RoxenLib class.

```
public static java.lang.String httpEncodeString(java
.lang.String f)
```

The following characters are replaced with % escapes: SP, TAB, LF, CR, %, ', ", NUL.

```
public static RoxenResponse httpLowAnswer(int erro
r,
                                         java.lang
.String data)
```

```
public static RoxenResponse httpLowAnswer(int erro
r)
```

```
public static RoxenResponse httpStringAnswer(java.l
ang.String text,
                                             java.l
ang.String type)
```

```
public static RoxenResponse httpStringAnswer(java.l
ang.String text)
```

```
public static RoxenResponse httpRXMLAnswer(java.l
ang.String text,
                                           java.lan
g.String type)
```

```
public static RoxenResponse httpRXMLAnswer(java.l
ang.String text)
```

```
public static RoxenResponse httpFileAnswer(java.io.R
eader text,
                                           java.lan
g.String type,
                                           long len
)
```

```
public static RoxenResponse httpFileAnswer(java.io.R
eader text,
                                           java.lan
g.String type)
```

```
public static RoxenResponse httpFileAnswer(java.io.R
eader text)
```

```
public static RoxenResponse httpFileAnswer(java.io.I
nputStream text,
                                           java.lan
g.String type,
                                           long len
)
```

```
public static RoxenResponse httpFileAnswer(java.io.I
nputStream text,
                                           java.lan
g.String type)
```

```
public static RoxenResponse httpFileAnswer(java.io.I
nputStream text)
```

```
public static RoxenResponse httpFileAnswer(java.io.F
ile text,
                                           java.lan
g.String type,
                                           long len
)
                                           throws java.io.
FileNotFoundException
```

```
public static RoxenResponse httpFileAnswer(java.io.F
ile text,
                                           java.lan
g.String type)
                                           throws java.io.
FileNotFoundException
```

```
public static RoxenResponse httpFileAnswer(java.io.F
ile text)
                                           throws java.io.
FileNotFoundException
```

```
public static RoxenResponse httpRedirect(java.net.UR
L url)
```

```
public static RoxenResponse httpAuthRequired(java.l
ang.String realm,
                                             java.l
ang.String message)
```

```
public static RoxenResponse httpAuthRequired(java.l
ang.String realm)
```

```
public static RoxenResponse httpProxyAuthRequired(j
ava.lang.String realm,
                                                  j
ava.lang.String message)
```

```
public static RoxenResponse httpProxyAuthRequired(j
ava.lang.String realm)
```

End of /roxen/2.1/programmer/java/HTTP.html

LastResortModule

Start of /roxen/2.1/programmer/java/LastResortModule.html

```
public RoxenResponse last_resort(RoxenRequest id)
```

End of /roxen/2.1/programmer/java/LastResortModule.html

LocationModule

Start of /roxen/2.1/programmer/java/LocationModule.html

The interface for modules which have a specific URL path in the virtual file system.

```
public java.lang.String queryLocation()
```

```
public RoxenResponse findFile(java.lang.String f,
                              RoxenRequest id)
```

```
public java.lang.String[] findDir(java.lang.String f
,
                                  RoxenRequest id)
```

```
public java.lang.String realFile(java.lang.String f,
                                 RoxenRequest id)
```

```
public int[] statFile(java.lang.String f,
                      RoxenRequest id)
```

End of /roxen/2.1/programmer/java/LocationModule.html

Module

Start of /roxen/2.1/programmer/java/Module.html

```
java.lang.Object
   |
   +--com.roxen.roxen.Module
```

The base class for Roxen modules. All modules must inherit this class, directly or indirectly.

Each module should also implement one or more of the specific module type interfaces.

```
public static final int TYPE_STRING
```

```
public static final int TYPE_FILE
```

```
public static final int TYPE_INT
```

```
public static final int TYPE_DIR
```

```
public static final int TYPE_STRING_LIST
```

```
public static final int TYPE_MULTIPLE_STRING
```

```
public static final int TYPE_INT_LIST
```

```
public static final int TYPE_MULTIPLE_INT
```

```
public static final int TYPE_FLAG
```

```
public static final int TYPE_TOGGLE
```

```
public static final int TYPE_DIR_LIST
```

```
public static final int TYPE_FILE_LIST
```

```
public static final int TYPE_LOCATION
```

```
public static final int TYPE_TEXT_FIELD
```

```
public static final int TYPE_TEXT
```

```
public static final int TYPE_PASSWORD
```

```
public static final int TYPE_FLOAT
```

```
public static final int TYPE_MODULE
```

```
public static final int TYPE_FONT
```

```
public static final int VAR_EXPERT
```

```
public static final int VAR_MORE
```

```
public static final int VAR_DEVELOPER
```

```
public static final int VAR_INITIAL
```

```
public Module()
```

```
public abstract java.lang.String queryName()
```

```
public abstract java.lang.String info()
```

```
public RoxenConfiguration myConfiguration()
```

```
protected java.lang.String queryInternalLocation()
```

```
protected RoxenResponse findInternal(java.lang.String
 f,
                                     RoxenRequest i
d)
```

```
public java.lang.String status()
```

```
protected void start()
```

```
protected void stop()
```

```
protected void defvar(java.lang.String var,
                      java.lang.Object value,
                      java.lang.String name,
                      int type,
                      java.lang.String doc)
```

```
protected void defvar(java.lang.String var,
```

```
                      java.lang.Object value,
                      java.lang.String name,
                      int type)
```

```
public java.lang.Object query(java.lang.String name
)
```

```
public int queryInt(java.lang.String name)
```

```
public java.lang.String queryString(java.lang.String
 name)
```

```
protected void set(java.lang.String name,
                    java.lang.Object value)
```

```
protected void set(java.lang.String name,
                    int value)
```

End of /roxen/2.1/programmer/java/Module.html

ParserModule

Start of /roxen/2.1/programmer/java/ParserModule.html

The interface for modules which define RXML tags.

```
public SimpleTagCaller[] querySimpleTagCallers()
```

End of /roxen/2.1/programmer/java/ParserModule.html

ProviderModule

Start of /roxen/2.1/programmer/java/ProviderModule.html

The interface for modules providing services to other modules.

```
public java.lang.String queryProvides()
```

End of /roxen/2.1/programmer/java/ProviderModule.html

RoxenClassLoader

Start of /roxen/2.1/programmer/java/RoxenClass-Loader.html

```
java.lang.Object
   |
   +--java.lang.ClassLoader
         |
         +--java.security.SecureClassLoader
               |
               +--java.net.URLClassLoader
                     |
```

```
                  +--com.roxen.roxen.RoxenClass-
Loader
```

```
public RoxenClassLoader(java.net.URL[] urls)
```

```
public void addJarFile(java.lang.String jarFileName)
              throws java.io.FileNotFoundExceptio
n,
                  java.io.IOException
```

```
public static java.lang.String getModuleClassName(
java.lang.String jarFileName)
                                          throws j
ava.io.FileNotFoundException,
                                              j
ava.io.IOException
```

End of /roxen/2.1/programmer/java/RoxenClass-Loader.html

RoxenConfiguration

Start of /roxen/2.1/programmer/java/RoxenConfigura-tion.html

```
java.lang.Object
   |
   +--com.roxen.roxen.RoxenConfiguration
```

A class representing the configuration of a virtual server in the Roxen server.

```
public RoxenConfiguration()
```

```
public java.lang.String getRealPath(java.lang.String
 filename,
                              RoxenRequest id
)
```

```
public java.lang.String getFileContents(java.lang.St
ring filename,
                              RoxenReques
t id)
```

```
public java.lang.String getMimeType(java.lang.Stri
ng filename)
```

```
public java.lang.Object query(java.lang.String name
)
```

```
public java.lang.String queryInternalLocation(Module
 m)
```

```
public java.lang.String queryString(java.lang.String
 name)
```

```
public java.lang.String queryInternalLocation()
```

```
public Module[] getProviders(java.lang.String provid
es)
```

```
public Module getProvider(java.lang.String provides)
```

End of /roxen/2.1/programmer/java/RoxenConfiguration.html

RoxenFileResponse

Start of /roxen/2.1/programmer/java/RoxenFileResponse.html

```
java.lang.Object
  |
  +--com.roxen.roxen.RoxenResponse
        |
        +--com.roxen.roxen.RoxenFileResponse
```

A class of responses using a file as their source. Use the methods in the HTTP class to create objects of this class.

End of /roxen/2.1/programmer/java/RoxenFileResponse.html

RoxenLib

Start of /roxen/2.1/programmer/java/RoxenLib.html

```
java.lang.Object
  |
  +--com.roxen.roxen.HTTP
        |
        +--com.roxen.roxen.RoxenLib
```

A support class containing useful methods for interpreting requests and synthesizing responses.

```
public static java.lang.String htmlEncodeString(jav
a.lang.String str)
```

The following characters are replaced with HTML entities: `&`, `<`, `>`, `"`, `'`, NUL.

```
public static java.lang.String htmlDecodeString(jav
a.lang.String str)
```

All HTML 4.0 entities are replaced with their literal equivalent.

```
public static java.lang.String doOutputTag(java.ut
il.Map args,
                              java.uti
l.Map[] varArr,
                              java.lan
g.String contents,
                              RoxenReq
uest id)
```

This method is used to create tags such as database query tags, where zero or more results in the form of variable bindings are applied to a fixed template, and the results of the subsitutions are concatenated to form the total result.

```
public static java.lang.String parseRXML(java.lan
g.String what,
                              RoxenReque
st id)
```

```
public static java.lang.String makeTagAttributes(ja
va.util.Map in)
```

```
public static java.lang.String makeTag(java.lang.S
tring s,
                              java.util.Ma
p in)
```

```
public static java.lang.String makeEmptyElemTag(j
ava.lang.String s,
                              jav
a.util.Map in)
```

```
public static java.lang.String makeContainer(java.l
ang.String s,
                              java.u
til.Map in,
                              java.l
ang.String contents)
```

End of /roxen/2.1/programmer/java/RoxenLib.html

RoxenRequest

Start of /roxen/2.1/programmer/java/RoxenRequest.html

```
java.lang.Object
  |
  +--com.roxen.roxen.RoxenRequest
```

A class representing requests from clients.

```
public final RoxenConfiguration conf
```

```
public final java.lang.String rawURL
```

```
public final java.lang.String prot
```

```
public final java.lang.String clientprot
```

```
public final java.lang.String method
```

```
public final java.lang.String realfile
```

```
public final java.lang.String virtfile
```

```
public final java.lang.String raw
```

```
public final java.lang.String query
```

```
public final java.lang.String notQuery
```

```
public final java.lang.String remoteaddr
```

```
public final long time
```

```
public final RoxenConfiguration configuration()
```

```
public java.util.Map variables()
```

```
public java.util.Map requestHeaders()
```

```
public java.util.Map cookies()
```

```
public java.util.Set supports()
```

```
public java.util.Set pragma()
```

```
public java.util.Set prestate()
```

```
public void cache(int sec)
```

```
public void noCache()
```

End of /roxen/2.1/programmer/java/RoxenRequest.html

RoxenResponse

Start of /roxen/2.1/programmer/java/RoxenResponse.html

```
java.lang.Object
  |
  +--com.roxen.roxen.RoxenResponse
```

The base class for response objects. Use the methods in the HTTP class to create response objects.

```
public void addHTTPHeader(java.lang.String name,
                          java.lang.String value)
```

End of /roxen/2.1/programmer/java/RoxenResponse.html

RoxenRXMLResponse

Start of /roxen/2.1/programmer/java/RoxenRXMLResponse.html

```
java.lang.Object
  |
  +--com.roxen.roxen.RoxenResponse
       |
       +--com.roxen.roxen.RoxenStringResponse
            |
            +--com.roxen.roxen.RoxenRXMLResponse
```

A class of responses using an RXML parsed string as their source. Use the methods in the HTTP class to create objects of this class.

End of /roxen/2.1/programmer/java/RoxenRXMLResponse.html

RoxenStringResponse

Start of /roxen/2.1/programmer/java/RoxenStringResponse.html

```
java.lang.Object
  |
  +--com.roxen.roxen.RoxenResponse
       |
       +--com.roxen.roxen.RoxenStringResponse
```

A class of responses using a string as their source. Use the methods in the HTTP class to create objects of this class.

End of /roxen/2.1/programmer/java/RoxenStringResponse.html

SecurityModule

Start of /roxen/2.1/programmer/java/SecurityModule.html

The interface for modules that should be marked as security modules in the module listing. It contains no methods, implementing it just flags the module as security related.

End of /roxen/2.1/programmer/java/SecurityModule.html

SimpleTagCaller

Start of /roxen/2.1/programmer/java/SimpleTagCaller.html

The interface for handling a single specific RXML tag

```
public static final int FLAG_NONE
```

```
public static final int FLAG_EMPTY_ELEMENT
```

```
public static final int FLAG_NO_PREFIX
```

```
public static final int FLAG_PROC_INSTR
```

```
public static final int FLAG_DONT_PREPARSE
```

```
public static final int FLAG_POSTPARSE
```

```
public static final int FLAG_STREAM_RESULT
```

```
public static final int FLAG_STREAM_CONTENT
```

Note: It might be obvious, but using streaming is significantly less effective than nonstreaming, so it should only be done when big delays are expected.

```
public static final int FLAG_STREAM
```

```
public static final int FLAG_DEBUG
```

```
public java.lang.String queryTagName()
```

```
public int queryTagFlags()
```

```
public java.lang.String tagCalled(java.lang.String tag,
                                  java.util.Map args,
                                  java.lang.String contents,
                                  RoxenRequest id,
                                  Frame frame)
```

End of /roxen/2.1/programmer/java/SimpleTagCaller.html

UniqueModule

Start of /roxen/2.1/programmer/java/UniqueModule.html

The interface for modules that may only have one copy in any given virtual server. It contains no methods, implementing it just prevents multiple copies of the module from being added to a virtual server.

Perl

The perl support, provided by the module of that same name, offers two ways of using perl with Roxen: running perl scripts and running in-line perl code. The script support caches scripts so they (and the perl interpreter) don't need to be reloaded every time the script runs. The in-line Perl code support allows running snippets of perl code inside `<perl>...</perl>` containers in RXML pages or within `<?perl ... ?>` processing instructions.

End of /roxen/2.1/programmer/perl/index.xml

Using In-Line Perl Code

Start of /roxen/2.1/programmer/perl/tag.xml

How to use the `<?perl ... ?>` processing instruction in your RXML pages.

In-line perl code in `<?perl ... ?>` processing instructions can be executed much like scripts, and the same subset of the Apache API is available as in the perl script support described in the API section.

Note that the default value for the "Perl Tag Support" configuration option is "No", and that this has to be changed to "Yes" for in-line Perl code to be executed. Also note that the "RXML-parse tag results" option has to be switched to "Yes" if you want the output produced by the Perl code to be RXML parsed afterwards.

An example perl PI tag:

```
<?perl
Roxen->request()->print(scalar localtime);
?>
```

End of /roxen/2.1/programmer/perl/tag.xml

Running Perl Scripts

Start of /roxen/2.1/programmer/perl/script.xml

With the "Script output" option of the perl support module set to HTTP, it is possible to run old perl CGI scripts more or less as they are, only with better performance, provided they don't rely on their environment being reset upon each run of the script. (This performance boost comes in part from the script already being loaded and compiled, thus staying resident in-between requests and in part from the fact that there is no need to fork off new processes for the script.)

The environment variables are the same as those available in standard CGI, plus those added by roxen for your convenience.

As with CGI, anything you print to STDERR ends up in the server's general debug log file (or the console, in the event of your starting the server with the `--once` flag).

CGI-Style Scripts

A simple example of a traditional CGI script in perl could look like:

```
#! /usr/local/bin/perl
print "Content-type: text/plain\r\n\r\n";
print "Environment variables:\n";
for (sort keys %ENV)
{
  print $_, "=", $ENV{$_}, "\n";
}
```

mod_perl-style Scripts

Turning this one into a mod_perl-style script, also runnable by the Perl support module and by some considered a bit tidier, might result in something along the lines of:

```
my $r = Roxen->request();
$r->print("Content-type: text/plain\r\n\r\n");
$r->print("Environment variables:\n");
for (sort keys %ENV)
{
  $r->print($_, "=", $ENV{$_}, "\n");
}
```

End of /roxen/2.1/programmer/perl/script.xml

Supported mod_perl API Methods

Start of /roxen/2.1/programmer/perl/api.xml

The Perl support for Roxen 2.1 has limited support for the Apache mod_perl interface. A request object can be obtained with

```
$r = Roxen->request();
```

or, for compatibility,

```
$r = Apache->request();
```

The request object functions currently supported include:

```
$r->print ( @list )
$r->printf( $formatstring, @list )
$r->status( [ $status ] )
$r->status_line( [ $status ] )
$r->method()
$r->uri()
$r->filename()
$r->protocol()
$r->log_error( $message )
$r->warn( $message )
$r->get_basic_auth_pw()
$r->get_remote_host()
$c = $r->connection()
```

The connection object supports the following functions:

```
$c->auth_type()
$c->user()
$c->remote_ip()
$c->remote_host()
```

Additionally, two auxiliary functions are available:

```
Roxen->unescape_url( $url )
Roxen->unescape_url_info( $url )
```

For compatibility, they are also available as

```
Apache->unescape_url( $url )
Apache->unescape_url_info( $url )
```

The standard Perl functions "print" and "exit" have been over-
loaded so that they should reasonably safe to use, but their use
is deprecated. Use $r->print() and $r->exit() instead.

End of /roxen/2.1/programmer/perl/api.xml

```
Apache->unescape_url( $url )
Apache->unescape_url_info( $url )
```

CGI

CGI - the freedom and possibly the shackles of free choice, and how CGI scripts can be put to use in the overall roxen picture.

CGI, the Common Gateway Interface (for an in-depth, external, non-roxen-specific reference, see a more official definition of the CGI specifications at http://hoohoo.ncsa.uiuc.edu/cgi/), is the age-old standard for making and running portable scripts on practically any web server. Roxen supports CGI via the CGI module.

The two good thing about CGI programming is that it works with any web server and that it allows the programmer full freedom of choice regarding the programming language. Unfortunately these are also the only benefits of CGI. For each request to a CGI script a program has to be run, something rather costly performancewise. CGI is not particularly easy to program; many complexities of web application programming must be handled by the CGI programmer. Nor are the security issues handled for you - it is entirely up to the programmer to take care about those issues herself.

Many of these shortcomings are however handled by languages and programming environments that use CGI to access the web server. With a good library, CGI programming can become easy for the programmer. It is however recommended to check how the library, language or environment handles the security implications of web application programming, and what the programmer needs to worry about.

Roxen makes it possible to integrate CGI programming with RXML. It is possible to embed calls to CGI scripts within RXML pages by using the `<cgi/>` or `<insert/>` tags. It is also possible for the RXML parser to post process output from CGI scripts. That way, a CGI script can make use of functionality from roxen modules.

The `<cgi/>` tag can be used together with the `<define>` tag to create new RXML tags that are handled via CGI scripts.

End of /roxen/2.1/programmer/cgi/index.xml

What is a CGI Script?

Start of /roxen/2.1/programmer/cgi/concept.xml

A CGI script is a program or script that is executed once for each request for it. The CGI script is either identified by file extension, for example `.cgi`, or by residing in a certain directory, for example `/cgi-bin/`. A request to a CGI script will be handled by finding the script and starting it with information about the request sent as environment variables and data on stdin. The script returns data by writing it to stdout.

The CGI script needs to be an executable file on the operating system. On Unix this is either a program, or a script that begins with `#!` followed by the name of the interpreter. On Windows this is either a program or a file with an extension bound to the suitable interpreter.

End of /roxen/2.1/programmer/cgi/concept.xml

Available Environment Variables

Start of /roxen/2.1/programmer/cgi/environment.xml

The use of environment variables for argument passing, as well as the handy roxen extensions to the standard.

A CGI script receives all of its parameters via environment variables. Since most of the variables received this way could be altered by a malicious cracker, it is a sound practice never to trust the content of these variables blindly and be sure to use proper quoting at all times. This reminder being said, here are the variables sent and their contents:

- When the browser authenticates itself, this variable contains the authentication type in use. The most common value is `Basic`.
- If any RoxenConfig configuration options were set, this is a space-separated array of them all.
- For requests which have attached information, such as HTTP POST and PUT, this is the length of the said content given by the client (available to read from stdin). This value may safely be trusted, since it is computed by roxen (and not fetched from the HTTP headers).
- This is the content type of the data provided; see CONTENT_LENGTH.
- A space-separated list of all cookie names sent with this request.
- The value of the cookie *name*.
- The filename of the CGI script.
- The path part of the URL to the CGI script.
- The version of the CGI protocol used, which is CGI/1.1 for Roxen 2.1.
- The contents of the HTTP Accept header of the request.
- The contents of the HTTP Accept-Charset header of the request.
- The contents of the HTTP Accept-Encoding header of the request.
- The contents of the HTTP Accept-Language header of the request.
- The contents of the HTTP Authorization header of the request. It will only be available if the `Raw user info` variable has been set to *Yes* by the server administrator.
- The net_loc part of the URL, as sent in the HTTP Host header, for instance `localhost:4711`.
- The contents of the HTTP Proxy-Connection header of the request.
- If the client sent any pragma header(s), they are provided here. For practical purposes, the only pragma header you're likely to encounter is `no-cache`, which some browsers (Netscape comes to mind) send when the user reloads the page.
- A lowercased version of the HTTP Connection header; typically `keep-alive`.
- The HTTP Referer header, if one was sent.
- The full name of the browser used.
- The query part of the URL, if present.
- The last modification date, HTTP formatted.

- If your server has the `Path Info Support` module loaded and the script was fed path info parameters, it is provided in this variable. The path info is the rest of the path segment in the URL following the pathname of your script.
- When the `PATH_INFO` variable is present, so is `PATH_TRANSLATED`. It contains a full filesystem path to the file that would be accessed when combining the directory name of the CGI script itself and the PATH_INFO variable.
- If roxen found prestates in the URL, this is a space-separated array of those present.
- The value `true` if that prestate was present.
- The query part of the URL, if present.
- The value of the form variable *name*.
- The IP number of the client machine.
- The domain name of the client machine, if roxen has had time to find it. Since it takes some time to find what domain name a computer has this information will not be available the first time a certain computer connects to the server.
- The port number used by the client.
- The login name used by the user.
- The password used by the user, only available if the `Send decoded password` variable is set to *Yes* by the administrator and the client authenticated itself.
- The method given in the HTTP request. In most cases, this will probably be GET or POST, but other HTTP methods, like PUT, are also possible. When using special protocols, such as WebDAV, other request methods may also occur.
- If the client authenticated itself ok with roxen's authentication module, this variable will contain the value `1`.
- If your server is setup to give all users user id cookies, this is the number of the user which requested the script right now.
- The complete path in the real file system to the CGI script.
- The path part in the URL.
- The domain name of the web server.
- The port number of the web servers. The default is 80 for HTTP or 443 for HTTPS, but it can be almost any value. If the server has several ports this variable will contain the port used to access the script.
- The protocol used.
- The name and version information of the web server.
- The URL to the web server. Together with SCRIPT_NAME, this makes up the complete URL for the script.
- A list of words, separated with spaces, of all features for which support information is available. See the supports chapter in the Website Creator manual for more information.
- The value `true` if that feature is supported by the current browser. See the supports chapter in the Website Creator manual for more information.
- A list of all form variables.
- The value of the form variable *name*.
- The value `true` if that RoxenConfig configuration option was set. See also `CONFIGS`.

End of /roxen/2.1/programmer/cgi/environment.xml

I/O Via the Standard Streams

Start of /roxen/2.1/programmer/cgi/streams.xml

The workings of the script's standard streams; stdin, stdout and stderr.

Stdin - Standard Input

Request methods such as HTTP POST and PUT have attached information in the request body that, unlike the common script parameters, is not present as environment variables. This information, CONTENT_LENGTH bytes in total, is instead available for reading from stdin. See also the previous page on environment variables.

Stdout - Standard Output

It is up to your script to write out whatever HTTP response headers it needs to stdout. After the headers (possibly none), the script must produce two newlines, and whatever is written after that ends up in the request body as the webpage seen by the client.

Stderr - Standard Error

Basically, your CGI script talks to the world through stdout and to the logs via stderr. By default, stderr is sent to the main server log, but this may be altered by the server administrator. Your debug and/or logging messages should typically end up on stderr. When your roxen process was started with the `--once` flag, all stderr output ends up in the console from which roxen was started.

An example script in `/bin/sh` exemplifying all of the above might look like this:

```
#!/bin/sh

if [ -z "$CONTENT_LENGTH" ]; then

cat << end_form
Content-type: text/html

<form method=post>
 <input type=text name=variable size=40
  value="Press submit to see a POST request body.">
 <input type=submit>
</form>
end_form

else

cat << end_reply
Content-type: text/plain

$CONTENT_LENGTH bytes of data were sent to us.
The content-type was $CONTENT_TYPE:

end_reply
cat

fi

echo "CGI example ran at `date`." 1>&2
```

End of /roxen/2.1/programmer/cgi/streams.xml