Roxen™ Programmer Manual

roxen
Challenger

www.roxen.com

Introduction

When creating web applications the web browser and web server provides a framework for the application. Development becomes a lot easier than creating client based GUI applications, since the programmer can take advantage of this framework. Any web application becomes truly platform independent, it can be accessed from any computer or appliance containing a web browser.

The web gives the programmer freedom to choose any programming language and environment. To the end user the application appears as a web page, what lies behind the scenes doesn't matter. This freedom means that almost every existing programming language have been used to make web applications, and that some scripting languages has been invented for the sole purpose of making web applications.

Roxen Challenger is of course an example of this, being written in Pike. Challenger is written to be extended, the modules that make up Challenger do not differ from modules made by third parties. But Challenger is not limited to be extended by modules; it supports standards like CGI and Java Servlets, to ensure that the programmer can chose the most suitable programming environment.

In fact Challenger makes it possible to integrate applications written in several languages and environments. Its own scripting language, RXML, is designed to make it possible to combine output from several applications or databases on each HTML page.

This programmers manual will describe the different ways to make web applications, and how they can be integrated to take full advantage of the power of Challenger.

Introduction This introduction chapter.

CGI and SSI How to use the CGI and SSI standards with Challenger.

Pike Tag How to use the `<pike>` tag, writing code directly in the web pages.

Pike scripts How to use Pike scripts.

Modules How to write Challenger specific modules.

Tag modules How to write modules that create their own RXML tags.

Location modules How to write location modules, such as file systems.

Other modules How to write other types of modules.

ID Object Information about the request id object.

Responses What responses can be sent from a module.

Library Methods Library Methods available within Challenger.

Databases How to connect to and use databases.

Appendix

Pike

One major difference between Challenger and other web servers is that Challenger is written in the same language, Pike, that is used for third-party extensions and scripts. Usually you write the server in a compiled language like C or C++ and then let third-party developers use another interpreted language like Perl, JavaScript or Visual Basic for extensions.

That Challenger is written in its extension language means a lot for how powerful extensions you can make. You use the same tools as the Challenger developers. The line blurs between making extensions to the server and developing the server itself. Since the whole server is delivered with source code you can find out how the Challenger developers solved problems.

There are four ways of making extensions written in Pike. CGI scripts can be written in Pike, as well as any other language. Pike scripts work like CGI scripts, but are handled internally in the server. The `<pike>` tag can be used to include Pike code within RXML pages. Finally there are modules, that use the same API as the modules distributed with Challenger. Modules have access to all the functionality available in Challenger and are installed and configured through the configure interface.

There are security implications when using Pike scripts, the `<pike>` tag or modules. Since they are run in the actual server process, rather than as an external process, your Pike code will have access to many internal data structures of Challenger. It is for example quite easy to shut down the server from within a module. Therefore the system administrator of a Challenger server must trust her Pike programmers.

As always when it comes to web programming it is essential to treat user input with suspicion. Even though you only wanted a word of data the user could send megabytes of machine code. Fortunately Pike makes it hard to make errors in handling handling user input. It is however very important to understand the issue, especially when making scripts that start external programs, write user input back to RXML pages or connect to databases.

RXML

RXML, RoXen Macro Language, is a set of tags that can be used within HTML pages. RXML in itself can be used to make it easier to design web pages, connect to databases or

even to create simple web applications. RXML is documented in the User manual.

RXML is designed to behave much like HTML, so that it becomes easy to learn for someone who has mastered HTML. Another property of RXML is that the actual tags are handled by different modules. Thus it becomes possible to create new modules that extend RXML with new tags.

This makes RXML perfect for making functionality available to HTML designers. Often the programmer is not the best person to actually do the HTML layout necessary for an application. With RXML the programmer can make the functionality of the application and let a designer embed in HTML pages that make up the layout.

The actual programming need not be made in Pike, although modules that handle RXML tags can only be written in Pike. It is possible to use the `<cgi>` tag to call on CHI scripts. Or a wrapper module could be written, calling on functions written in another programming language. It is especially easy to call on functions written in Java.

CGI and SSI

CGI, Common Gateway Interface, is a standard for executing scripts from a web server. Every major web server supports it and it is the only way to make script portable between web servers. Challenger supports CGI scripts via the *CGI executable support* module.

CGI addresses few of the security implications of programming on the web. The programmer of the CGI script has to deal with them herself. Thus CGI scripts have become a security problem. It is easy to find, download and install CGI scripts that may have security problems. Care has to be taken when designing and testing the CGI script so it will be capable of handling any user input. On the web you never know when someone will try giving your little script a few megabytes of machine code as input.

Challenger makes it possible to combine CGI programming with the unique functionality present in Challenger. Either the output of CGI scripts can be parsed by the RXML parser. Or the CGI script itself can be invoked from within RXML with the `<cgi>` tag. It is even possible to define new tags handled via CGI scripts by combining the `<define>` tag with the `<cgi>` tag.

Java

Challenger supports programming in Java by supporting Java Servlets as well as making it possible to call on Java modules from Pike. For the Java support to work Java JDK 1.2 must be installed and Pike must have been compiled with Java support, as explained in the Servlets page in the Administrator manual.

Java within Challenger takes advantage of the tight integration between Pike and Java. The Pike interpreter will start and control the Java interpreter through Java native method invocation. Both Pike and Java is run within the same process, in different threads. This integration means that the penalty for calling a methods in Java from Pike is extremely low. It becomes possible to mix the languages without worrying about performance.

The main use for this tight Pike/Java integration in Challenger is to write modules in Java. Every Challenger module needs to be written in Pike, but by making a small wrapper module in Pike that calls methods in Java the effect will be that of a modules written in Java.

CGI and SSI

CGI and SSI

CGI, common gateway interface, and SSI, server side includes, are two standards for running scripts by a web server, that work with with practically any web server. Challenger support CGI with the *CGI executable support* module and SSI with the *Main RXML Parser* module.

The good thing about CGI programming is that it works with any web server. Unfortunately this is the only good thing about CGI and SSI. For each request to a CGI script a program has to be run, something rather costly of performance. 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, the programmer has to take care about them 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.

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

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

CGI

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 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.

Environment Variables

When invoking CGI scripts Challenger passes a number of environment variables to the script.

AUTH_TYPE contains the authentication type in use. The most common value is "Basic".

COOKIES lists all cookie names associated with this request.

COOKIE_*name* contains the value of the cookie *name*.

DOCUMENT_NAME contains the name of the CGI script.

DOCUMENT_URI contains the path part or the URL to the CGI script.

GATEWAY_INTERFACE contains the version of the CGI protocol used, which will be CGI/1.1 for the 1.3 version of Challenger.

HTTP_ACCEPT contains the contents of the accept header of HTTP.

HTTP_ACCEPT_CHARSET contains the contents of the accept-charset header of HTTP.

HTTP_ACCEPT_ENCODING contains the contents of the accept-encoding header of HTTP.

HTTP_ACCEPT_LANGUAGE contains the contents of the accept-language header of HTTP.

HTTP_AUTHORIZATION contains the contents of the auth header of HTTP. It will only be available of the *Raw user info* variable has been set to *Yes*.

HTTP_USER_AGENT contains the name of the browser used.

INDEX contains the query part of the URL.

QUERY_STRING contains the query part of the URL.

QUERY_*name* contains the value of the form variable *name*.

REMOTE_ADDR contains the IP number of the client machine.

REMOTE_HOST contains the domain name of the client machine, if Challenger 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.

REMOTE_PORT contains the port number used by the client.

REMOTE_USER the login name used by the user.

REMOTE_PASSWORD the password used by the user, only available if the *Send decoded password* variable is set to *Yes*.

REQUEST_METHOD contains 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 possi-

ble. When using special protocols, such as WebDAV, other request methods may also occur.

SCRIPT_FILENAME contains the complete path in the file system to the CGI script.

SCRIPT_NAME contains the path part in the URL.

SERVER_NAME contains the domain name of the web server.

SERVER_PORT contains 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 seveal ports this variable will contain the port used to access the script.

SERVER_PROTOCOL contains the protocol used.

SERVER_SOFTWARE contains the version information of the web server.

SERVER_URL contains the URL to the web server. Together with SCRIPT_NAME, this makes up the URL for the script.

SUPPORTS contains a list of words, separated with spaces, of all features for which support information is available. See the supports chapter in the User manual for more information.

SUPPORTS_*feature* contains the value *true* if that feature is supported by the current browser. See the supports chapter in the User manual for more information.

VARIABLES contains a list of all form variables.

VAR_*name* contains the value of the form variable *name*.

<!--#exec-->

The SSI `<!--#exec-->` is documented in the SSI chapter of the Web Site Creator manual.

<pike> tag

The <pike> tag makes it possible to include Pike code within RXML pages. It is great for making advanced scripting that is not possible with the normal RXML tags. It does however have security implications, since you can do as much damage with the <pike> tag as with a module or Pike script. This tag should therefore be restricted to trusted users only.

The <pike> tag is available from the *Pike Tag Module*.

Using the <pike> tag

Using the Pike tag is simple:

```
<pike>
 Pike code
</pike>
```

The code contained in the <pike> tag is ordinary Pike code, and is executed as if it were a method. The <pike> tag will be replaced by its output. The output is always a string and is either returned or accumulated with calls to the output() method.

The output from the tag will always be parsed by the RXML parser. If the output() method is used each call to that method will be parsed separately.

The <pike> will inherit roxenlib and does thus have access to all library methods. There are also a few predefined variables:

args A mapping containing the attributes given to the <pike> tag.

id The id object of the current request.

defines All the items created with the <define> tag.

Example

```
<gtext><pike>
 output( "Hello visitor from " );
 output( id->remoteaddr );
</pike></gtext>
```

Pike script

Pike scripts are similar to CGI scripts, in so far that they consist of a file that is executed when the user tries to access it. Pike scripts are however handled differently than CGI scripts. Instead of starting an external program the Pike script is run internally by Challenger.

The way that Pike scripts are handled is much more efficient than starting external CGI scripts, Pike scripts will generally respond faster and use less resources. It is also possible for them to cache data between requests.

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.

Pike scripts are handled by the *Pike script support* module.

Using pike scripts

A Pike script is essentially a Pike file containing at least a `parse()`. The first time a Pike script is requested Challenger will compile it and call its `parse()` method. Subsequent requests will just have to call the `parse()` method of the already loaded script.

Pike scripts are reloaded by requesting it with the pragma no-cache header. This is usually achieved by pressing reload, or shift reload, in the web browser. The behavior can be changed by the `no_reload()` method.

A pike script should usually inherit `roxenlib`, to get access to all library methods.

The methods available to a Pike script are:

string|mapping|object parse(object id) The `parse()` is called once for each request. It returns either a string containing RXML code, a response mapping created via one of the response methods or a file object. The response methods are documented in the Responses chapter.

The id argument is the id object of the request. The id object is documented in the Id object chapter.

int no_reload(object id) The `no_reload()` is called when the script is invoked via a pragma no-cache request. The script will only be reloaded if the `no_reload()` returns 0. This method exists to ensure that only the administrator or programmer can reload a script, in case this is an expensive operation. The usual implementation will check in the id object for certain conditions when the script should be reloaded, for example depending on the ip number of the computer the browser is run on.

Example

```
inherit "roxenlib";

string parse( object id )
{
 return "<html>\n<head>\n"
 "<title>Simple Pike Script Example</title>\n"
 "</head>\n"
 "<body>\n"
 "You gave the following index argument: "
 "<gtext>\n"
 + html_encode_string( id->variables->index )
 + "</gtext></body></html>";
}
```

The Example script will return the value of the form variable index, rendered by the `<gtext>` tag. Note the call to the `html_encode_string()`, this is to prevent anyone from sending dangerous RXML code in the form variable. It is always necessary to perform such quoting when handling user input together with RXML.

Modules

The functionality of the Roxen Challenger web server is implemented as modules. There are no difference between a module delivered with Roxen Challenger and a module made by a third-party. All parts of the Roxen API are available to any module.

A module is a `.pike` file, with appropriate methods implementing the module API. It must contain contain a `register_module()` method, that returns information about the module such as what type of module it is. The module type determines all further interactions between Challenger and the module.

Almost all modules contain configuration variables, that can be changed in Challenger's configuration interface. These variables are defined by calling the `defvar()` method, usually from the constructor (the `create()` method).

In order to get the necessary prototypes and methods for the Roxen API, modules needs to include `module.h` and inherit `module`. Most modules also inherit `roxenlib` to get access to the library methods. Thus most modules begin with:

```
#include <module.h>

inherit "module";
inherit "roxenlib";
```

Loading and reloading

Modules are loaded by adding them to a virtual server, with the *Add Module* button in the configuration interface. Once a module has been enabled and configured it will by loaded automatically when the server restarts.

If you have created or installed a new module you will need to do a reload on the *Add Module* page, in order to get it to show up. If there are any compilation errors the module will not show up, and the errors will be reported in the *Event Log* or the debug log file.

Modules are identified by their file name only. Thus you may not call your module the same as an existing module, even if it is placed in a different directory.

If you have changed the a module and want to try the new version you need to reload it. This is done by pressing the *Reload Module* button on the appropriate module in the configuration interface.

In case there were any compilation errors they will show up directly when you try to reload the module. The reload will fail and the old version of the module will keep running while you fix the errors in the new version.

API methods

The API methods that are available to be called from the module are:

defvar() is used to register a configuration variable. It is usually called from the constructor (`create()` method).

query() returns the value of a configuration variable.

set() sets the value of a configuration variable.

query_internal_location() gives the path to this modules own unique internal mount path.

API methods that the module can implements are:

checkvar() checks the intended value of a configuration variable when the user changes it. If the value is not valid an error message can be returned.

info() shall return a the description of the module, which then overrides the description given in the *register_module* module.

query_name() shall return the name of the module, which then overrides the name given in the *register_module* module.

register_module() returns information about the module, such as which type of module it is. *register_module* must be implemented in all modules.

start() is called when the module is started, each time a configuration variable is changed as well as when the module is unloaded.

status() returns a status message that will be displayed in the configuration interface.

stop() is called when the module is disabled or reloaded as well as when the server is restarted or shut down.

find_internal() is called when a file is requested from this modules internal mount path.

Errors

Since Pike is an interpreted language a programming error cannot crash the program. Instead the Pike interpreter catches and handles errors. Challenger takes advantage of this by providing error messages with a full Pike backtrace, making it easy to pinpoint the problem. The error will be reported sent to the browser as well as being written in the debug and event log.

That error messages, complete with a Pike backtrace, are written to the browser is great for programmers debugging their applications. It might however scare rather than help the average user of a web site. Therefore it is recommended to turn of such error messages on production servers, by setting the *Global Variables/Show the internals* variable to *No*.

Module types

The module type determines how the module should interact with Challenger. Modules of different types will need to

implement different API methods. A single module can be of several types, if it implements all necessary API methods.

Each request processed by Challenger will usually result in calls to several modules. Challenger will go through all module types, and for each module type it will call the module with the highest priority, given that the trustlevel, security patterns and mountpoint allows access to that module. If the module returns that it could not handle the request Challenger continues with the module of the same type with the next highest priority, or moves to the next module type if there are no modules left.

If the module did handle the request different things happen depending on the module type and the return value. For some return values the request is finished and will be returned to the browser as is. For other values the processing will continue, the value will be sent as input to modules of other types.

The most common module types are *Location* and *Parser*. *Location* modules handle requests to a mountpoint of the virtual file system. *Location* modules are in that respect similar to Pike or CGI scripts, with the difference that the URL to a location module is set in the configuration interface.

Parser modules handle one or several RXML tags. It often makes sense to make the functionality of a module available as a RXML tag. This way it will be easy for users to integrate the functionality into their HTML pages.

The module types are:

Authentication modules handle authentication of users as well as providing user information. The information provided by an authentication module can be used in .htaccess files, modules security patterns or other modules. Authentication modules usually connect to a source of user information, such as NIS or LDAP databases. There can only be one Authentication module loaded in e virtual server.

First try modules are called before all other module types, except for authentication modules. This is used to catch certain types of requests before they begin being processed. One usage is to access to the web server from certain IP addresses.

URL modules rewrite URLs. They receive the current URL and may return another. This is useful when a web page is published under several names, when a web page has moved or for IP less virtual hosting.

Extension modules handle virtual file extensions. They get called before location modules.

Location modules handle accesses to a certain mountpoint in the virtual file system. They are used to implement access to file systems as well as for making modules that behave like Pike or CGI scripts. Location modules can be used to provide access to files stored in other places than the normal file system, for example a CVS repository. Note that several location modules can be asked to handle a request. For modules that have the same priority the one with the longest matching mountpoint will be called first.

File extension modules deal with files with a certain extension, such as `.html` or `.gif`. This may be utilized to decode compressed files before sending them to the user, turning a data file format into something more humanly readable, or just setting the correct file content type. File extension modules are called if a previous module returned a file object.

Parser modules define one or more RXML tags. These are the basis for all RXML functionality in Roxen, and enable the administrator to add new tags to the RXML language just by loading a new module.

Directory modules deal with directory listings and index files. If the requested resource is a directory, the directory module gets called. It will usually try to find a suitable index file, or create a directory listing by using the API to ask all Location modules handling that mountpoint for directory contents. There can be only one Directory module in a virtual server.

Filter modules work on the almost-ready data, just before it is sent back to the requesting browser. This may be used, as the name suggests, to filter out parts of the data that should not be sent.

Last try modules are called when all other modules have failed to produce anything at all from the request. Uses include producing nicer error messages as well as redirecting the request to other servers.

Log modules collect log information, and are called at the same time as the request leaves the server. A log module can be used to supplement or replace the built in log functionality.

create()

```
void create( object conf )
```

`create()` is the constructor that is always called when an object is created in Pike. For modules the constructor will be called with a configuration object as argument. That object contains the configuration for the virtual server, that might be needed for advanced modules.

Usually the constructor is used to define configuration variables with the `defvar()` method.

defvar()

```
void defvar( string varname,
 mixed value,
 string name,
 int type,
 string|void documentation,
 void|mixed misc,
 void|int|function(void:int) do_not_show )
```

The `defvar()` method is called by the module to register a configuration variable. It is usually called from the constructor (`create()` method).

varname is the name used to identify the variable, for example when accessing it with the `query()` method. Must not begin with a dash (-).

value the variable's default value. Its type depends of course on the *type* parameter.

name the name which will appear in the configuration interface. If the name contains a colon the variable will be shown in a submenu in the configuration interface. The part before the colon will become the name of the submenu, the part after the name shown for this variable.

type the variable's type. See below for the available types.

documentation a string documenting this variable.

misc it has meaning only for some variable types, as discussed in more detail below. If the variable has not one of those types you can set it to 0 or ignore it completely.

do_not_show is either an integer value or a function. If it is 0 the variable will be shown, if non-zero it will be hidden. The same applies to the return value of the supplied function. The most common use is to hide or show some variables depending on the value of another variable.

Variable Types

TYPE_FLAG is used to express a true/false variable. The user will be prompted with a yes/no choice, while internally an int will be used to express it.

TYPE_INT the variable is a integer number.

TYPE_STRING the variable is a string.

TYPE_FILE the variable is a path to a file in the real file system. It is stored internally as a string.

TYPE_DIR the variable is a path to a directory in the real file system. A slash is automatically appended at the end, if not already present, when a new value is set. It is stored internally as a string.

TYPE_DIR_LIST is a list of directories, stored as an array of strings. Each element is handled in the same way as a TYPE_DIR variable.

TYPE_INT_LIST is an array of integers.

TYPE_STRING_LIST is an array of strings.

TYPE_MULTIPLE_INT the variable will contain an integer, chosen from a choices list. The *misc* argument is used to supply the list, as an array of integers.

TYPE_MULTIPLE_STRING the variable will contain a string, chosen from a choices list. The *misc* argument is used to supply the list, as an array of strings.

TYPE_LOCATION the variable is a mountpoint in Challenger's virtual file system. It is handled internally as a string.

TYPE_COLOR the variable is an integer containing a 24-bit RGB color value, coded as `red << 16 + green <<8 + blur`. The user will be offered a more friendly input format.

TYPE_TEXT the variable is a string, possibly containing multiple lines of input.

TYPE_PASSWORD the variable is a password, that will be handled as a string run through unix `crypt()`. The password will thus not be stored in clear text.

TYPE_FLOAT the variable is a floating point number.

query()

```
mixed query( string varname )
```

The `query()` method is used to access configuration variables. *varname* is the name of the variable, as defined by the `defvar()`. The return value is the value of the variable. Querying an undefined variable will result in an exception.

set()

```
void set( string varname, mixed value )
```

The `set()` method is used to set the value of a configuration variable from within the module. *varname* is the name of the variable, *value* its new value.

query_internal_location()

```
string query_internal_location()
```

Returns the internal mount point of this module. Requests to this mount point will be sent to the `find_internal()` method.

check_variable()

```
void|string check_variable( string name, mixed
value );
```

The `check_variable()` method is called when the user changes a configuration variable. If 0 is returned the variable will be changed to the new value. If a string containing an error message is returned the variable will not be changes, and the error message will be displayed in the configuration interface.

The *name* argument contains the name of the configuration variable while the *value* argument contains the intended new value of it.

info()

```
string info()
```

The `info()` returns a string with the documentation of this module. It will override the documentation set by the `register_module()` method and will be shown in the configuration interface.

query_name()

```
string query_name()
```

Returns the name of the module, as shown in the configuration interface. This method will override the information defined by the `register_module()`.

register_module()

```
array register_module()
```

The `register_module()` returns an array describing the module. The array shall have the following contents:

```
({ module_type, name, description, 0,
only_one_copy })
```

module_type is an integer with a bitfield of the module's type. The type is obtained either by using one type constant or by using the bitwise or operator ('|') on several. The module type constants are; MODULE_EXTENSION, MODULE_LOCATION, MODULE_URL, MODULE_FILE_EXTENSION, MODULE_PARSER, MODULE_LAST, MODULE_FIRST, MODULE_AUTH, MODULE_DIRECTORIES, MODULE_LOGGER and MODULE_FILTER.

ditem>name is a string containing the name of the module. This can later be overridden by implementing a `name()` method.

description is a string containing a longer description of the module, that will be shown in the configuration interface. It can later be overridden by implementing an `info()` method.

only_one_copy determines if it should be possible to have more than one copy of the module per virtual server. If zero it will be possible to have several copies. If set to another value it will only be possible to have one copy per virtual server.

start()

```
void start( int count, object conf )
```

The `start()` method is called when a module is loaded, as well as when a configuration variable has been changed and are being saved. In the first case the *count* argument will be zero, in the later it will be two. The *conf* argument is the configuration object for the virtual server.

status()

```
string status()
```

Called by the configuration interface to let the module tell the user about it's internal status. This method must return a string containing HTML code that can be fitted in the `<dd>` part of a `<dl>` list.

If this method is implemented the module will show *Status and debug info* in the configuration interface.

stop()

```
void stop()
```

The `stop()` is called when the module is disabled or reloaded as well as when the web server is restarted or shut down. It makes it possible for a module to go down nicely.

find_internal()

```
mixed find_internal(string path, object id)
```

The `find_internal()` method is called when a file is fetched from this modules internal mount path. All modules have an unique mount path that can be used if the module needs to serve some files, but it doesn't matter what URL those files get. The *Graphic text* module does for example use an internal mount point for the images it has generated. Modules that has no need for an internal mount point does not need to implement this method.

The return value of this method is either a mapping created with one of the response methods, see the responses chapter, or a `Stdio.File` object containing the requested file.

All URLs to files within the internal mount point should be generated by the module itself. It is necessary to call the `query_internal_location()` method to find out where the internal mount point is located.

Parser modules

A parser modules handles one or several RXML tags. RXML tags comes in two flavors, plain tags or container tags. The difference is that container tags require an ending tag, and encloses content. `<img>` is an example of a plain tag while `<h1>...</h1>` is an example of a container tag.

A parser module works by first register tags or container tags via the `register_tag_callers()` and/or `register_container_callers()` methods. The tags are registered with name and callback method. Later, when the tag are discovered by the *Main RXML parser*, its callback method will be called.

The module type constant for parser modules is MODULE_PARSER.

query_tag_callers()

`void|mapping(string:function) query_tag_callers()`

The `query_tag_callers()` method is called by Challenger to find all plain tags handled by the module. It returns a mapping of string, function pairs where the string is the name of the tag and the function is a callback method that handles that tag. The method need not be implemented if a particular parser module has no plain tags but only container tags.

The definition for the acctual callback method is:

`void|string|array(string) tag_caller( string tag, mapping (string:string) att, object id)`

tag contains the name of the tag being parsed. It makes it possible to have one tag caller that handles several tags slightly differently. *att* contains the attributes sent to the tag. *id* is the request information object.

Attributes are decoded and sent as a mapping with the attribute name as key and the attribute value as value. Attributes that does not have a value is given their name as value. Thus `<tag hi=hopp foo>` will be decoded as `([ "hi" : "hopp", "foo" : "foo" ])`. It won't be possible to separate from `<tag hi=hopp foo=foo>`.

The return value will usually be a string that will replace the tag. The string will in turn will be RXML parsed. Thus a tag caller can return RXML that will be parsed. Care must however be taken to quote things properly before returning them, otherwise it might be possible for a user to get his input RXML parsed.

If zero is returned the tag will be left as is. If an array containing one string is returned the tag will be replaced with that string, but the string won't be RXML parsed.

query_container_callers()

`>void|mapping(string:function) query_container_callers()`

The `query_tag_callers()` method is called by Challenger to find all plain tags handled by the module. It returns a mapping of string, function pairs where the string is the name of the tag and the function is a callback method that handles that tag. The method need not be implemented if a particular parser module has no plain tags but only container tags.

The definition for the actual callback method is:

`void|string|array(string) container_caller( string tag, mapping (string:string) att, string contents, object id)`

tag contains the name of the tag being parsed. It makes it possible to have one tag caller that handles several tags slightly differently. *att* contains the attributes sent to the tag. *contents* contains the contents that were enclosed between the start and end tags. *id* is the request information object.

Thus `<tag hi>Hello</tag>` will be called with `({ "hi" : "hi" })` as att and "Hello" as contents.

The return value will usually be a string that will replace the tag. The string will in turn will be RXML parsed. Thus a tag caller can return RXML that will be parsed. Care must however be taken to quote things properly before returning them, otherwise it might be possible for a user to get his input RXML parsed.

If zero is returned the tag will be left as is. If an array containing one string is returned the tag will be replaced with that string, but the string won't be RXML parsed.

Location modules

Location modules handle a mountpoint in Challengers virtual file system. Request to a URL under a modules mountpoint will be sent to that module. Unless there are more location modules with overlapping mountpoints, in which case the module with the longest mountpoint will be tried first. So if there are one module mounted on `/test/` and another on `/` the module mounted on `/test/` will be tried first. If it returns that it could not handle the request the module mounted on `/` will be tried.

Location modules are either Pike or CGI scripts written as modules or file systems. For script like modules it will only be necessary to implement the `find_file()` method. The advantages of writing scripts as modules is that they can be configured and installed nicely.

Location modules that implement file systems need to implement all API methods. This is to make it possible to interact fully with other module types, like directory modules. File system modules usually either give access to the normal file system, but do some special processing or access control, or give access to files stored somewhere else, for example in a CVS repository. For the first type of file system modules it can be a good idea to inherit the *Filesystem* module.

The module type for the Location Modules is `MODULE_LOCATION`.

API methods

find_dir() returns a directory listing. It is used indirectly by directory type modules to create directory listings.

find_file() is the fundamental method of all location modules. It will be called to handle all accesses to the modules mount point.

query_location() returns the mountpoint of the module. It can be omitted if the module has a configuration variable *location*.

real_file() returns the path to a file in the real file system. It can of course only be implemented if the file in question exists in the real file system.

stat_file() returns information about a file, in the same format as Pike's `file_stat()` method.

find_dir()

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

The `find_dir()` gives a directory listing; an array of strings containing the names of all files and directories in this directory. *path* is the path to the directory, in the modules 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()` will probably be used.

find_file()

```
mixed find_file(string path, object id)
```

The `find_file()` method is the fundamental method of a location module, that all location modules need to implement. It is called when a request is made for an URL within the modules mount point. *path* contains the path to the object, in the modules name space. *id* 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 modules mount point. If a module is mounted on `/test/` and a user requests `http://my.server/test/files/img/hej.gif` the module will be called with a path of `files/img/hej.gif`. That way the administrator can set the mount point to anything she wants, and the module will keep working.

If the module could not find the requested object the return value is zero. In that case Challenger will move on and try to find in in other location modules. If the requested object is a directory the return value is minus one, 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 mapping created with one of the response methods, see the responses chapter, or a `Stdio.File` object containing the requested file.

query_location()

```
string query_location()
```

The `query_location()` returns a string containing the mount point of this module, in Challenger's virtual file system. However it is rarely necessary to implement, since the mount point can be obtained automatically from a configuration variable named *location* or type TYPE_LOCATION. The mount point should almost always be configurable by the administrator anyway.

real_file()

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

The `real_file()` method translates the path of a file in the modules name space to the path to the file in the real file system. *path* is the path to the file in the modules 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.

stat_file()

```
void|array(int) stat_file(string path, object
id)
```

The `stat_file()` emulates Pike's `file_stat()` method, returning information about a file or directory. *path* is the path to the file or directory in the modules 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 an array of integers in the following format:

```
({ 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 director, minus three that it is a symbolic link and minus four that it is a special device. This value must be given.

atime is an integer containing the last time the file was accessed, as seconds from 1970. It can be ignored.

mtime is an integer containing the last time the file was modified, as seconds from 1970. It will be used to handle Last-Modified-Since requests and should be supplied if possible.

ctime is an integer containing the time the file was created, as seconds from 1970. It can be ignored.

uid is an integer containing the user id of the 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.

Other module types

This chapter gives describes the module API for the less commonly used module types.

Authentification

An authentification type module are used to verify user authentification and provide user information. The most common use is to provide a connection to an existing user database, for example a NIS or LDAP. There can only be one authentification module loaded in a virtual server.

The module type constant is MODULE_AUTH.

The special methods for Authentication Modules are:

array auth(string authdata, object id) is called with the authentification data as sent by the browser and should return an array suitable for further processing by Challenger. The *authdata* array has the format; `({ "Basic", basic_auth_data })`. In the future there may be other authentification schemes that *basic* in use, in which case the array contents may change.

The *basic_auth_data* contains a string with user name and password, separated by colon.

The `auth()` method should return an array of the following format; `({ successp, username, password })`. *authp* is either one, for successful authentification, or zero if the authentification was unsuccessful. If the authentification was successful the *password*

The `auth` method should return an array whose first element is 1 if the authentication was successful, and otherwise 0. The second element should be the user name. The third is either 0 (for successful authentication) or a string containing the invalid password (for failed authentication).

array userinfo(string user_name) fetches information about a certain user. *user_name* contains the login name of the user. It should return an array of the following format: `({ user_name, password, uid, gid, real_name, home_directory, login_shell })`.

user_name is a string containing the user's login name. *password* is the password of the user, usually encrypted. It need not be present at all. *uid* is an integer containing the user id. *gid* is an integer containing the user's primary group id. *real_name* is a string containing the real name of the user. *home_directory* is the path to the users home directory. It is used by the *User filesystem* module to provide access to users' home pages. Finally *login_shell* contains the login shell used by the user. It is used by the ftp protocol to emulate the behavior of the normal unix ftpd.

array user_from_uid(int uid) this is another method that fetches information about users, but it uses the user id as key rather than the login name. The method returns an array of the same type as the `userinfo()` method.

array userlist() returns an array with the names of all users in the user databases. For performance reasons some authentification modules will not allow this but rather return an empty array.

Directory

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.

The following API methods are available:

mixed parse_directory(object id) returns a normal response containing either a suitable directory listing or an index file. The path to the directory is found in `id->not_query`.

Extension

An extension module handle a virtual file extension. It will be called before any location modules, in case the use requests an URL ending with that extension.

The module type constant is MODULE_EXTENSION.

The API methods are:

array (string) query_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.

mixed handle_extension(string extension, object id) is the method that will be called to do the actual work. *extension* is the extension of the request, *id* the request information object. For possible return values see the responses chapter.

File extension

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.

The available API methods are:

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.

mixed handle_file_extensions(Stdio.File file, string extension, object id) is the method that will be called to do the actual work. *file* is the file object that a previous module returned. *extension* is the extension of the request, *id* the request information object. For possible return values see the responses chapter.

Filter

Filter modules are called for every request, just before the request leaves Challenger. The module type constant is MODULE_FILTER.

mixed filter(mapping response, object id) The *response* argument contains the response Challenger were about to send to the browser, *id* contains the request information object. If the filter module returns zero the original response will be sent. If the filter module returns something else, that response will be sent to the browser instead.

First

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.

The available API method is:

mixed first_try(object id) *id* is the request information object. For possible return values see the responses chapter.

Last

A last module is called in case no other module could handle the request. Its module type constant is MODULE_LAST. Its API method is:

mixed last_resort(object id) *id* is the request information object. For possible return values see the responses chapter.

Log

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. The available API method is:

void|int log(object 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, no other log modules will be called nor will the internal logging take place.

Provider

Provider modules are modules that provide services to other modules. The module type constant is MODULE_PROVIDER. The available API method is:

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:

object 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.

array(object) 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, mixed ... 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, mixed ... 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.

URL

URL modules are called after first modules. URL modules change data in the request object, for example the URL being fetched. The module type constant is MODULE_URL. The available API method is:

void|object|mixed remap_url(object id) *id* is the request information object. `remap_url()` either returns

zero, a changed request information object or a normal response, as documented in the response chapter. If the method returns a request information object Challenger will call all URL modules again, so care must be taken so infinite loops are not created.

Request information object

The request information object contains information about the actual request being processed. The header information sent by the browser is present, as well as information added by various modules. Since the request information object is sent to every module and callback method involved in handling the request it is a perfect place for intra-module communication.

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

The members of the request information object are:

conf virtual server configuration object. This field is not necessarily available when running ordinary Pike scripts.

variables a mapping containing all form variables sent in the request, as well as any additional variables created by modules or RXML tags.

supports a multiset containing the features the current browser supports. See the supports chapter in the creator manual for more information.

pragma contains a multiset with information sent by the HTTP header Pragma. *no-cache* is a important value, since that mean that the request should be fetched without caching.

id->misc a mapping available to store miscellaneous information. Usually used for intra-module communication or communication between different tags in the same module. Beware that the name space of this mapping is getting very cluttered, take care to choose names that will likely remain unique before storing things here.

raw a string containing the the entire raw client request.

query a string containing the query part of the URL. It is usually easier to get information from the *variables* mapping.

not_query a string containing the path part of the URL.

raw_url a string containing the whole URL. Note that usually the URL as seen by the web server does only contain the path and query part. Full URLs are only used for proxy requests.

auth an array containing authentification information. If an authentification module is present it will contain an array of the format; `({ successp, username, password })` where successp is one if the user succeeded in authentificating and zero otherwise. The password will not be available if the user succeeded in authentificating herself.

If there are no authentification module present *auth* will contain an array of the format; `({ "Basic", basic_auth_info })` where basic auth info contains a string with the user name and password, separated by colon.

remoteaddr a string containing the numeric IP address of the client machine.

clientprot a string containing the protocol version used by the client when issuing the request.

method a string containing the access method specified by the client. This is usually GET, or sometimes POST when forms are being used. It can also have other values if you allow methods like PUT, user ftp or use special protocols such as WebDAV, may try to use various other methods as well.

request_headers a mapping containing the HTTP headers that the client submitted when issuing the request.

my_fd is a Stdio.File object containing the actual network connection to the browser.

Responses

Many API methods have a common set of responses, that are:

zero means that the module could not handle the request.

minus one means that the requested object was a directory. The request will be sent to a directory module, if present.

Stdio.File the file object will be sent to the browser, after the *Content types* module has determined the appropriate content type.

response mapping contains all information necessary for Challenger to send the result to the browser. It includes header information as well as file content. The response mapping should not be created by hand but rather by an appropriate response method.

The response methods are available if *roxenlib* has been inherited. They are:

http_string_answer(string contents, void|string type) simply returns the *contents* as the content type *type*, or by default `text/html`.

http_rxml_answer(string rxml, object id, void|object(Stdio.File) file, string|void type) returns *rxml* after sending it through the RXML parser, as `text/html` unless the *type* argument is given.

http_file_answer(Stdio.File file, void|string type, void|string len) returns the contents of *file* which should be open for reading.

http_auth_required(string realm, string message) is used to prompt the user to log on. A web browser will open a dialog prompting the user to fill in her user name and password. The *realm* argument is a string which will be used to distinguish different protected domains on the same server from each other. The *message* argument will be shown if the user decides not to try to log on.

http_redirect(string url, void|object id) creates a redirect response that will make the web browser try to fetch the redirected page. *id* is only required if the URL is a relative URL, that is, one that doesn't specify a protocol and server, in which case the `http_redirect()` will need more data to create a complete URL.

http_pipe_in_progress() tells Challenger that your module will take charge of delivering data to the user. Challenger will ignore the request from now on and let the module handle the file object associated with the request. The file object is found in the `my_fd` field of the id object.

Library methods

This chapter gives an overview of the various methods available to a module that has inherited *roxenlib*.

string html_encode_string(string s) convert a string to HTML by quoting all characters that have special meaning in HTML. Should always be used when inserting user input into RXML code.

string html_decode_string(string s) convert an HTML string to a plain string by unquoting HTML escape sequences.

string do_output_tag(mapping args, array (mapping) var_array, string contents, object id) is used to process output tags, like `<sqloutput>`. *att* is the attributes sent to the tag, they are necessary since there are certain attributes that affect all output tag. *var_array* contains an array of mappings with string, string pairs. `do_output_tag()` will loop over the array and insert the variables in the mappings in appropriate places in the *contents*.

string make_tag(string tag, mapping att) creates a HTML tag with the name *tag* and the attributes from *att*.

string make_container(string tag, mapping att, string contents) creates a HTML container tag, with the name *tag* the attributes *att* and the contents *contents*.

string parse_rxml(string rxml, object id); runs the RXML parser on the string *rxml*.