Masser - administrator's and developer's guide

written by Jakub Spicak <jakub.spicak@deltaes.cz> 1.0 ($Revision: 1.2 $), $Date: 2003/10/07 08:07:27 $

Masser is an environment for creating web applications based on perl language and fast-cgi technology (see www.fastcgi.com). It's prepared for database connection and created for maximum efficiency.

Contents

1 Installation

1.1 Requirements

System requirements:

- UNIX type operating system (Linux, Solaris and HP-UX recommended)
- Perl 5.005+
- web server with fastcgi module (apache 1.3.26+ recommended)

Perl modules required:

- Delta a DeltaX
- FCGI 0.65+

1.2 Installation - apache

Mod_fastcgi is supposed to be installed and functional. Now you can:

- copy masser.fcgi file into some directory in which it will be visible for apache (for example DOCUMENT_ROOT/masser)
- copy img/ directory and files in it into some directory in which it will be visible for apache too (for example DOCUMENT_ROOT/masser/img)
- set ExecCGI for directory where masser.fcgi is (Options +ExecCGI)
- file masser.conf copy to one of these places:
  - directory with masser.fcgi (not recommended)
  - /etc
  - /usr/etc

  and adjust these settings:

  - path - according to place (or places) where applications and their configurations will be (for example /usr/local/lib/masser)
  - img - according to your place of img/ directory (absolute or relative URL to masser.fcgi) - /masser/img in this example
  - app_path - according to your place of masser.fcgi (URL without protocol part) - server_name/masser/masser.fcgi in this example
  - other settings (see next chapters for explanation)

- if you want to use masser as a static fast-cgi application (which is recommended), add to your httpd.conf this line:

```
FastCgiServer /path/to/masser/masser.fcgi -processes 1
```

  (number of processes as you need)

1.3 Application installation

Application will probably consist of these directories (or at least some of them):

- etc/ with configuration file skeleton - copy it to some of directories from path option in masser.conf or config/ subdirectory in it
- app/ with application itself - copy it in some directory from path option in masser.conf, in subdirectory named by application (APP for example)
- img/ with images for application - it must be placed into directory with masser images, subdirectory named by application (DOCUMENT_ROOT/masser/img/APP)
- other depends on application (sql/ with SQL scripts etc.)
- adjust application configuration and run application:

```
http://server/path/masser.fcgi/APP
```

2 How it works?

Main engine is in masser.fcgi file, which must be installed in such a place to be accessible by web server. Applications themselves can be placed anywhere, but I recommend to store them outside of DOCUMENT_ROOT of web server not to be viewed using web server due to error or bad configuration.

It's fast-cgi application, so main program is started by web server (at start or at first request - regarding configuration).

2.1 Startup initialization

Immediately after start program will read configuration:

- if environment variable `MASSER_CONFIG` is set, configuration file is read from file set in this variable
- otherwise configuration is read from `masser.conf` file
- if this setting is not absolute path, masser is trying to find the file in these directories:
  - current directory
  - directory `/usr/etc`
  - directory `/etc`

2.2 Request processing

When a request is received, system processes parameters, first of all from request URL, then from request itself (GET or POST type).

1. after URL part leading to script *masser* one or two (or three in **debug mode**) parts are processed:
   - if the first argument is `_source`, program will show source code of the page
   - first part is supposed to be application code
   - second part is assumed to be (list of) page(s) requested to be shown

   Examples:

   `/path_to_masser/masser/TEST`
   : will show (default) page of TEST application

   `/path_to_masser/masser/_source/TEST/page1`
   : will show source code of page page1 from TEST application

2. next following variables are worked out (if they are not set from previous point):

   APP
   : application name (uppercase is recommended), default is *APP*

   PG
   : list of pages to be shown (comma separated), default is *default*

   OP
   : list of operations to be done (comma separated), default is none (empty)

   LANG
   : language code, default is *CZ*

   SCH
   : schema code, to be used by application

   SID
   : Session Identification, to be used by application

Next application is initialized (if not used before):

- read and parse configuration according to masser settings, filename is assumed to be in this form: *application name converted to lowercase with .conf suffix*
- database connect initialization (regarding configuration)
- read and parse files with labels in languages according to configuration, filenames are assumed in <app>_lang.<lang_code> form (app always in lowercase) and masser tries to find it for every path from masser.conf and app.conf this way:
  - directory / app_name / `lang` / file
  - directory / app_name / file
  - directory / `lang` / file
  - directory / file
- application initialization: masser tries to find file <app>.init (app converted to lowercase) like this:
  - directory / app_name / `init` / file
  - directory / app_name / file
  - directory / `init` / file
  - directory / file

  If this file is found, it's compiled and run *only once* (for every masser instance).[1] This code is intended to actions to be performed only once in "application life" (for example read some specific configuration, setting some special database parameters etc.). Persistent variable *%p* can be used - read more in some next chapter.

Then all requested actions and pages will run.

Action or page is processed this way:

- compilation if not compiled yet[2]
- check or connect required databases
- prepare SQL statements if they are not prepared yet[3]
- set correct values to *%txt* hash (according to **LANG** variable)
- write source (compiled) code, if *write_code_dir* is set
- set persistent hash *%p* values
- run the code
- check pending transactions, write error if there are any and rollback them (regard *pending_transaction_rollback* setting)

[1] This is not true for debug mode, in this mode init action is run every request.
[2] always in debug mode
[3] always in debug mode

3 Configuration

3.1 masser.conf

`max_cycles`

number of requests; after processing them masser will be restarted (exits and fast-cgi will start it again), recommended value is between 100 - 2000

`path`

list of directories, where masser will find application configurations and files (semicolon separated)

`img`

image directory (URL, can be relative to masser.fcgi)

`log_file`

full path and name of main masser log

`log_error`

where to write errors:

- file - to file set in `log_file`
- stderr - to stderr
- file,stderr - both file and stderr

`log_warn`

where to write warnings

`log_info`

where to write information messages

`log_debug`

where to write debug messages

`debug`

if set, masser is running in debug mode

`content-type`

Content-Type setting for generated pages (used in `page_header` function)

`app_path`

URL to masser itself, without protocol specification (http or https is added automatically), for example `ip_or_name_of_server/path/masser.fcgi`

`access_log`

full path with filename, where information about accesses to masser will be written. If not set, no log will be created.

`access_log_format`

access_log format, you can use following variables:

- *%app* - application name
- *%pg* - page
- *%op* - action
- *%ip* - remote IP address
- *%date* - date and time of request

`performance_log`

full path with filename, where information about masser performance will be written. If not set, no log will be created.

`performance_log_format`

performance_log format, you can use following variables:

- *%app* - application name
- *%pg* - page
- *%op* - action
- *%ip* - remote IP address
- *%date* - date and time of request
- *%dtime* - time (in seconds) needed to process the request
- *%dmem* - memory increase (in bytes) while processing the request[4]
- *%mem1* - size of memory allocated (in bytes) before processing request[5]
- *%mem2* - size of memory allocated (in bytes) after processing request[6]

`max_memory_size`

maximal memory size allocated by masser; if masser is bigger than this value, it will exit to release memory. Checkout is performed after processing each request.[7]

`pending_transaction_rollback`

masser checks if there are no started transactions after processing all actions and pages in request (in active database connections). If there are any, it writes error to log and - if this parameter is set - performs rollback.

3.2 <app>.conf

Application configuration files.

`path`

this value will be added to `path` from `masser.conf` only for this application, other applications will not be influenced

[4] at this time works only on Linux
[5] at this time works only on Linux
[6] at this time works only on Linux
[7] at this time works only on Linux

`languages`

list of languages in which application texts exist (comma separated), uppercase is recommended

`log_file`

full path and filename of log file of application

`log_error`

where to write errors:

- file - to file set in `log_file`
- stderr - to stderr
- file,stderr - both file and stderr

`log_warn`

where to write warnings

`log_info`

where to write information messages

`log_debug`

where to write debug messages

`dbX_*`

db connection settings, X is number from 1 to 5:

- dbX_driver - database driver (see DeltaX::Database)
- dbX_name - database name
- dbX_user - database user
- dbX_passw - database user password
- dbX_datestyle - date and time format (see DeltaX::Database)
- dbX_trace - trace level (see DeltaX::Database)
- dbX_stat - statistics level; see db_stat
- dbX_statfile - file to write statistics; see db_statfile

`write_code_dir`

if set, masser will write to given directory every (compiled) source code (both action and page) before it tries to run it; name is composed like this: <app>_<type>_<name>.code, where <app> is name of application, <type> is type (pg - page, ac - action) and <name> is object name

`db_trace`

database trace level for DeltaX::Database, this is used if no dbX_trace is set

`db_stat`

statistics level for DeltaX::Database, this is used if no dbX_stat is set; syntax is: `db_stat = type[,max_high[,max_all]]`, where:

- type is statistics level:

  - none - no statistics (default)
  - sums - only count of performed statements, errors and total time required to perform them
  - high - sums + top statements
  - all - high + all statements
- max_high - count of top statements to collect and write; default is 5
- max_all - count of all statements to collect and write; default is 100

`db_statfile`

where to write statistics (full path with filename), this is used if no dbX_statfile is set

Application configuration file can have any other parameters, masser ignores them and they are for application use.

4 Syntax

4.1 Pages (.pg)

Pages are main part of application and most of other files are related to them. According to `PG` variable value masser will try to find file to process regarding `path` from masser and application configuration. It tries to find file according to PG value with .pg suffix (searches first in paths from application and then from masser):

- directory / application / `pages` / file
- directory / application / file
- directory / `pages` / file
- directory / file

Basic parts of pages are tags `<?` and `?>`. Code between these tags is processed, other code will be copied to output (see DeltaX::Page for details).

You can use variants of these tags which are described in next sections.

Code in page is local; it means that function or variable in one page is not visible from other pages (in the same or other application), except of global variables and functions.

Example - application APP, file page1.pg:

```
...
    <?
        sub my_func { print 'page1'; }

        my_func();
    ?>
...
```

Example - application APP, file page2.pg:

```
...
    <?
        sub my_func { print 'page2'; }

        my_func();
    ?>
...
```

In page page1 string "page1" will be written, "page2" will be written in page page2.

4.1.1 Code <? ... ?>

Code between these two tags is supposed to be perl code and is without changes processed. Developer can assume existing of these global variables:

- **%g** - global hash - universal use, is reset after each masser cycle (all actions and pages from one request are processed)
- **$app** - application
- **$lang** - selected language
- **$sch** - schema identification
- **$sid** - Session ID
- **$query** - CGI object[8]

Next you can use following abbreviations which will be replaced:

- `PAR(id)` - value from application configuration
- `TXT(id)` - text from actual language file
- `G(id)` - is replaced by `$g{'id'}`

4.1.2 Print value <?= ... ?>

This variant is used for inserting value to actual place in page. Code <?=$variable?> is equivalent to <? print $variable; ?>.

Example:

```
...
    <h1> <?='TXT(app_title)'?> </h1>
...
```

[8] You don't need to use this variable, all functions from CGI module are present, see `use CGI qw(:all);`

4.1.3 Comment <?!---- ... ----?>

Comment will be ignored by masser and will not be even copied to output, it's completely thrown.[9]

4.1.4 Include <?!include ...?>

Syntax: <?!include filename [def1[,def2...]]?>

Masser tries to find file with given name similar to pages (.pg):

- directory / application / `include` / file
- directory / application / file
- directory / `include` / file
- directory / file

Code from the file is included into parent file, so you can use any construction from it (include is used often in pages, so you can write to it code which can be written to page itself).[10] DefX definitions can be used to control included parts of .inc file using if, else and end constructs.

Example:

```
File test.pg:
  <?!include table print?>
File table.inc:
  <?!use table?>
  <?!javascript table?>
  <?:if print?>
    <?!css table_print?>
  <?:else?>
    <?!css table_screen?>
  <?:end?>
```

4.1.5 Package <?!package ...?>

Syntax: <?!package filename?>

Masser tries to find file [filename].pkg similar to pages (.pg) this way:

- directory / application / `package` / file
- directory / application / file
- directory / `package` / file
- directory / file

[9] Opposite to HTML comment (<!---- ... ---->), which will be copied to output.
[10] Masser currently not checks for recursive including, so it can lead to indefinite cycling of it!

It works similar to *include*, but is intended to create so-called packages - codes which consist of some parts (for example JavaScript code, CSS and module).

Example:

```
<?!include button.inc?>
<?javascript button?>
<?!css button?>
```

DefX definitions can be used to control included parts of .pkg file using if, else and end constructs.

Example:

```
File test.pg:
  <?!package table print?>
File table.pkg:
  <?!use table?>
  <?!javascript table?>
  <?:if print?>
    <?!css table_print?>
  <?:else?>
    <?!css table_screen?>
  <?:end?>
```

4.1.6 Module <?!use ...?>

Syntax: <?!use filename

Masser tries to find file [filename].pm this way:

- directory / application / `modules` / file
- directory / application / file
- directory / `modules` / file
- directory / file

This directive is used for including module to page. It can be placed anywhere in page, it's processed during compilation. Module which can be "used" using this directive is very similar to classic perl module, but you can use some masser features - read more about it in one of the following chapters.

4.1.7 Page beginning <?!page_start ...?>

Syntax: <?!page_start [key1=>value1, key2=>value2, ...]?>

Writes start of HTML page (HTTP header and start of page to BODY tag). You can use any parameter which is recognized by `start_html()` function from CGI module and these special:

- **title** - page title (default is "Page Title")

- **do_form** - if set (and do_start is set too), prints start of form with hidden variables for masser (PG, OP, etc.). Default is true.
- **do_start** - if set, prints start of page (using start_html()), default is true.
- **on-load** - this value is used for on-load action in BODY tag (not recommended), default is none.
- **content** - Content-Type for HTTP header, default is content-type from masser.conf
- **cookie** - cookies, default is none, not used unless do_start is set

[11]

Example:

```
...
    <?!page_start title=>'TXT(app_title)', bgcolor=>'#ffffff'?>
...
```

Important: This function includes JavaScript code and CSS to page, so if you don't use it, tags `javascript` and `css` won't work!

4.1.8 Page ending <?!page_end?>

Syntax: <?!page_end?>

This directive writes end of page - end of FORM, BODY and HTML tags. It should be used as the last output of the page.

4.1.9 JavaScript include <?!javascript ...?>

Syntax: <?!javascript [ext:]filename?>

It requests that given JavaScript code to be included into page output. Code is normally written to HEAD tag of the page, but if use ext: prefix in the name, masser will include only reference to a file, otherwise it tries to find the file **[filename].js** this way:

- directory / application / `javascript` / file
- directory / application / file
- directory / `javascript` / file
- directory / file

Its content is read and included into the page.

In JavaScript files you can use following constructions:[12]

[11] This directive must be the first output of page, because it writes HTTP header. In other case server writes Internal Error = bad header.

[12] You cannot use it in files included with `ext:` prefix - they are not processed.

- `PAR([id])` is replaced by content of $conf{'id'} (configuration)
- `TXT([id])` is replaced by content of $txt{'id'} (text)
- `GLB([id])` is replaced by content of $g{'id'} (global hash)

Example - file example1.js:

```
alert('TXT(error_required)');
var num_items = PAR(max_results);
for (var i=0; i<G(nusers); i++) { alert(i); }
```

Example - page:

```
<?!javascript example1?>
```

This directive can be placed anywhere in the page or in any included file, it's processed during compilation.

4.1.10 CSS include <?!css ...?>

Syntax: <?!css [ext:]filename?>

It requests that given file (it's content) to be included into page code (into HEAD tag). You can use ext: prefix to say to masser to not include code, only reference.[13] Otherwise masser tries to find a file **[filename].css** this way (there is small difference in opposite to other findings: $schema [SCH parameter in page] is used to enable schema switching):

- directory / application / `css` / `param('SCH')` / file
- directory / application / `css` / file
- directory / application / file
- directory / `css` / `param('SCH')` / file
- directory / `css` / file
- directory / file

File is read and included into the page.

In CSS files you can use following constructions:[14]

- `PAR([id])` is replaced by content of $conf{'id'} (configuration)
- `TXT([id])` is replaced by content of $txt{'id'} (text)
- `GLB([id])` is replaced by content of $g{'id'} (global hash)

[13] Because of CGI module limitation, you cannot insert more than one CSS file with ext: prefix. If you include more than one, only last will be included.

[14] You cannot use it in ext: prefixed files - they are not processed.

Example - file example2.css:

```
BODY { background-image: url(GLB(img_dir)/backg1.gif); }
```

Example - page:

```
<?!css example2?>
```

This directive can be placed anywhere in the page or in any included file, it's processed during compilation.

4.1.11 Prepare SQL command <?!statement ...?>

Syntax: <?!statement [dbnum:]filename?>

Prepares given SQL command (using open_statement - see DeltaX::Database). Using dbnum prefix (number from 1 to 5) you can set in which database connection it is to be prepared (if application uses more than one database connection). If not set, statement is prepared in connection number 1.

Masser searches for file **[filename].sql** this way:

- directory / application / `statement` / file
- directory / application / file
- directory / `statement` / file
- directory / file

You can use following constructions in it:

- `_DATE_` inserts constructions for date
- `_DATETIME_` inserts construction for date and time

Example - file MY_STAT.sql:

```
SELECT * FROM cats_users WHERE pernr = ? AND last_chgp = _DATETIME_
```

Example - page:

```
    <?!statement MY_STAT?>
    <?!statement 2:OTHER_STAT?>
...
    <?
        my $result = $db->perform_statement('MY_STAT');
    ?>
```

Prepared statements are global for the whole application, but do not influence other running applications. If masser is not in debug mode, each statement is prepared only once, even it's included in more pages. As JavaScript or CSS it can be placed anywhere in code, it's processed during compilation.

4.1.12 Function call <?!do ...?>

Syntax: <?!do filename [parameters]?>

At given place calls - differently from include it will not include whole code - given function, which code is in file **filename**. Masser tries to find it this way:

- directory / application / `function` / file
- directory / application / file
- directory / `function` / file
- directory / file

It's content must be perl code, you can use PAR, TXT and G replacements and global variables.

Function is global for the whole application, it's invisible for other running applications.

Example - file func.pl:

```
my (undef,$par1,$par2) = @_;
# WARNING! The first argument is always application name!

if ($par1 > $par2) {
    G(to_print) = 'TXT(is_greater)';
} elsif ($par1 < par2) {
    G(to_print) = 'TXT(is_lesser)';
} else {
    G(to_print) = 'TXT(is_equal)';
}
```

Example - page:

```
...
    <?!do func.pl 5,8?>
    <?=$g{'to_print'}?>
...
```

4.2 Actions (.ac)

Actions are called according to `OP` variable, which can be empty (no action will be performed), one or more actions (separated by commas).

All actions are performed **before** pages.

Masser tries to find action according to OP with .ac suffix this way:

- directory / application / `actions` / file
- directory / application / file

- directory / `actions` / file
- directory / file

Syntax of these files is similar to pages, but in practice it includes only perl code and `include` and `statement` directives. It can output text, but it's not recommended.

5 How to write...

5.1 ...pages

In pages (.pg) you can use directives as described in previous chapters. But you can use following variables, too:

%g

Global hash which is often used for transporting values between action and page. Be careful, its validity is limited by one masser cycle (which means that it's cleared after processing all actions and pages in one request).

So you cannot use it to remember values between two requests (you cannot know which masser will process request if there is running more than one). You must use HTML and/or HTTP ways to do it (hidden fields, cookies). See persistent hash described in next text.

While coding you can use G(item) form, so `print G(name);` and `print $g{'name'};` are equal.

%p

Global hash. Main difference from %g is its persistence between requests. You can use it mainly for data filled to it in *init* action or to remember some value between two masser cycles (request process).

Be careful, this is very limited in case masser is running as more than one instance (fast-cgi technology enables this and it's very useful). No one can say you, what masser instance (process) will get next request. ***This hash is persistent only in one instance of masser!***

Be careful while using this variable, it's not intended to store large pieces of data.

You cannot use it to send information between two applications even in one masser instance, every application has its own hash.

%txt

This hash contains texts loaded from language file according to selected language (LANG variable). *Read only (changes will be dropped after processing request).*

%conf

This hash contains configuration read from <app>.conf. *Read only (changes will be dropped after processing request).*

$app

This variable contains actual application (as param('APP')).

$pg

This variable contains actually running page (as param('PG')).

$lang

This variable contains actually selected language (as param('LANG'));

$db, $db1, $db2, $db3, $db4, $db5

These variables contain DeltaX::Database objects numbered according to application configuration (see dbX_* parameters). Note: $db = $db1.

You can use following functions in your code:

`error, warn, info, debug, trace`

Functions from DeltaX::Trace, used for printing messages to application log.

`break_page`

This function allows you to change page, which is processed by masser (value of $pg and param('PG'), but you cannot do this by changing these values).

The only parameter of this function is new page name. Masser then does this:

- if the new page is the same as actual page, nothing is done
- if not, variables which control actually shown page are changed
- *after actual page is processed* given page is run

It's used for example for switching to another page (for example error showing page) in case insufficient grants are detected or so.

Remember that everything, what was printed till calling this function, cannot (and is not) be undone.

Other useful notes:

- code can be conditionally included this way:

```
...
<? if ($something) { ?>
    <?!include file1.inc>
<? } else { ?>
    <?!include file2.inc>
<? } ?>
```

But using this code, contents of both files are included. You cannot control compile-time instructions (like include) by run-time variables...

- you can interrupt page processing using **return**, **die** (will be written into application log as error), or by jump to end of page **goto END_THIS_PAGE;**. Yes, final code, into which pages are compiled in masser, is anonymous function ;-).

5.2 ...actions

You can use everything (except break_page) as in pages here (see previous chapter). I only want to recommend here to use actions for doing some work and pages for printing output. It's not technical but logical difference. There are situations in which it probably cannot be done, but at least try to do it.

5.3 ...modules

Modules for masser are in fact same as "classic" perl modules. There are only two differences: You can place it outside perl lib directory (but masser must find it) and you can use some variables from masser (they are *%conf, %txt, %g, %p, $app, $pg, $lang, $db, $dbX*) this way:

```
package xy;
...
use vars qw/%txt %conf $app/;
import main;
...
```

5.4 ...JavaScript and CSS

I only want to discuss here using GLB(...) for inserting information from global hash %g: You must remember that this replacement is done in function **page_start**, so this hash (or at least keys used in javascript or css files) must be filled before calling page_start() (so calling function which fills it, include, module or code which fills it) must be placed before its calling.