dokeos®
open source e-learning

Brussels, July 2, 2007

Dokeos Administration Level II Training

dokeos®
open source e-learning

Table of contents

Level assessment

Discussing previous level achieved

There have been a few months since the last training about Dokeos administration was given. This section is made to review a few concepts you have seen in the your Dokeos Administration Training Level I and make sure they have been remembered so that you can fully benefit from the current training.

Principles to remind

- setting up an Ubuntu server (installing, next, next, next, finish, ok, coffee, good!)
- the super-administrator is called *root*, as is the root of all directories */,* but you cannot login as *root* as on most other UNIX/Linux systems (you have to use *sudo*)
- security is never absolute
- getting the admin permissions using *sudo* (a major difference compared to other Linux systems)
- setting up the *Apache* web server
- setting up the *MySQL* database
- setting up *PHP5* and Dokeos' required extensions (GD, PEAR – to get PECL and the dynamic upload bar, MySQL, ...) with *apt-get install* or *synaptic packages manager*
- setting up an *Apache Virtual Host* (to use a name rather than an IP address, harder to remember for human beings)
- installing Dokeos and setting the permissions on Dokeos' directories

Training focus

As you realise, you have already been trained to the basic principles of administering a Dokeos system. The current training will not only focus on the way Dokeos works (giving you the ability to make it work better), but also on how the other tools work together to provide a reliable, powerful and fast learning application server.

For sure, the fact that you requested this training must mean that you've already asked yourself some questions about what you were going to learn in this documentation and training. So why not take a few minutes to discuss *your* main focus with the trainer, so that he knows what has to be seen in more detail?

Setting up the test server

Introduction

This section will take us through the whole setup of a test server that we are going to use along this training session, to make sure we practice what we just learnt. Setting up the server will involve using a pre-installed local Linux computer and make sure everything is ready, checking it step-by-step before installing Dokeos.

Plan enough accesses (web, SFTP, shell, ...)

The first step to working together on one computer is to ensure that everything you need to connect is there.

To connect to another computer, we recommend using secure channels. SSH is a secure protocol but is sometimes a bit difficult, if not necessary, to use on the command line. SFTP is a protocol based on SSH that works the same as FTP, so you can send and download files in a good security. Most decently recent FTP clients also support it (ex.: FileZilla).

Make sure you create an account on the computer to configure (a different account from root) and then connect and try the *sudo* command to see if you are authorised to do some admin activities on this server.

Servers

We need a web server (*Apache*) and a database server (*MySQL)*. Checking they are installed on our computer is simple: check it graphically, using the Synaptic Package Manager, or check it on the command line, using the *dpkg -l* command to list all the packages installed.

If one of the necessary packages is not installed, you can use the ***Synaptic Packages Manager*** again to install it, or you can use the ***apt-get install*** command to do the same.

The packages you need are:

- apache2-mpm-prefork : Web server
- mysql-server-5.0 : Database server
- libapache2-mod-php5 : PHP5
- mysql-client-5.0 : Database client
- phpmyadmin : a graphical client for MySQL

Once you are sure that you have everything you need, let's configure a *virtual host* and install Dokeos! There is an excellent installation guide available on the website.

Code

PHP

PHP is an easy-to-learn language. You can learn as much as you want about PHP on the PHP website: http://www.php.net/ and follow the links to documentation and tutorials, but we are going to see it briefly here.

Basically, PHP is an interpreted language, mostly used to write scripts for the web, so as long as you have a web server running and the PHP module configured, the web server will "interpret" your script and transform the code into useful data for the user. For example, printing the current time would simply be one file containing this line

```
<?php  print(date('Y-m-d'));?>
```

The

```
<?php
```

part is called an *opening tag*.

The

```
print(...)
```

is a function (recognized by the parenthesis following it) that prints something to the web page.

The

```
date(...)
```

is a function (again, recognized by the parenthesis following it) that returns a string of characters representing a date, using a specific format. Here, we use ***Y-m-d*** to print the Year (in four digits), followed by the **m**onth (two digits), followed by the **d**ay (two digits).

The

```
 ;
```

marks the end of a command, or *statement* in PHP terminology.

The

```
?>
```

part is called a *closing tag.*

This gives us a pretty good idea of how PHP scripts must be formed. You can also mix it with HTML to print a static page with only a little dynamic part. You type some code on the server, and the user gets a dynamic result in his browser. And this is how Dokeos works.

Code structure and location

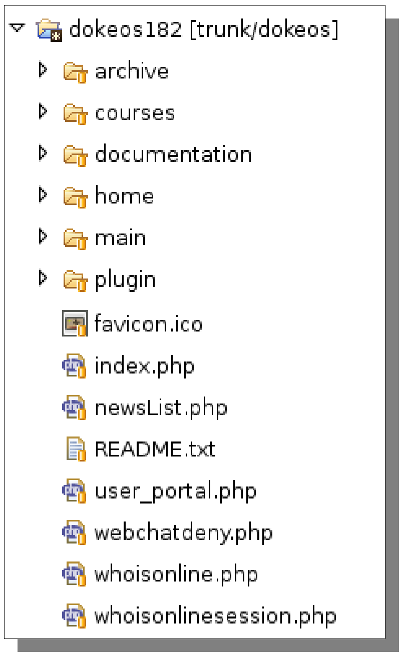
The Dokeos code structure is as complex as the number of features is high. The Dokeos code has an already long history and a lot of people have worked on it, which gives it an unusual structure which could certainly be improved (and we are actively working on this).

You will find all the Dokeos code in the Dokeos root directory, which we will call *dokeos/* for the purpose of shortening the term in the future. Specifically, you will find only a few scripts in that *dokeos/* : the *index.php*, the *user_portal.php*, and the *whoisonline.php* scripts.

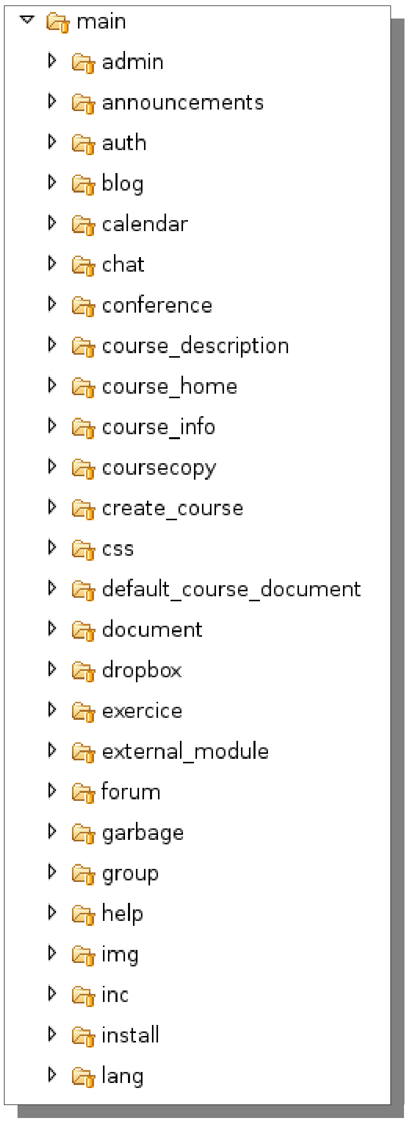
index.php is the public script everybody gets to when accessing the portal for the first time

user_portal.php is the script that displays your list of courses and your menus

whoisonline.php is the script that lets you know who is online



```
▽ dokeos182 [trunk/dokeos]
  ▷ archive
  ▷ courses
  ▷ documentation
  ▷ home
  ▷ main
  ▷ plugin
    favicon.ico
    index.php
    newsList.php
    README.txt
    user_portal.php
    webchatdeny.php
    whoisonline.php
    whoisonlinesession.php
```



```
▽ main
  ▷ admin
  ▷ announcements
  ▷ auth
  ▷ blog
  ▷ calendar
  ▷ chat
  ▷ conference
  ▷ course_description
  ▷ course_home
  ▷ course_info
  ▷ coursecopy
  ▷ create_course
  ▷ css
  ▷ default_course_document
  ▷ document
  ▷ dropbox
  ▷ exercice
  ▷ external_module
  ▷ forum
  ▷ garbage
  ▷ group
  ▷ help
  ▷ img
  ▷ inc
  ▷ install
  ▷ lang
```

Each of these scripts, and most of the other Dokeos scripts, include very important configuration and function library scripts.

All the configuration scripts lie in *dokeos/main/inc/conf/*

Most of the libraries lie in *dokeos/main/inc/lib/*

If you have a look in *dokeos/main/*, you will easily find that all the course's tools have one directory of their own. This goes for the dropbox, the documents, the learnpath (*newscorm*) and the *exercise* tool, for example.

Each of these directories contains the PHP code specific to a tool.

The *admin/* directory contains the code for the administration section.

tracking/ contains the code for the users tracking.

img/ contains all Dokeos' icons

css/ contains the web styles (*Cascading Style Sheets*)

In the root directory (*dokeos/*), you will find a *courses/* directory, which contains all documents of all courses. This is the directory that is the most likely to change in size on your portal.

In an admin perspective, you should also pay attention to *dokeos/main/garbage/* and *dokeos/archive/*, which might contain old files that need manual (or automatic) cleaning.

The *dokeos/main/upload/users/* directory is one more directory to keep an eye on. Users can upload their pictures there.

Analysing PHP performance

There are several ways to analyse PHP performance. They range from the quick and punctual way to the long and systematic way. You can:

- use manual testing – watch a page and see that it is too slow and that something should be done
- use small commands in the script – use microtime() to get the current time at the beginning and at the end of the script, and print the difference so you know how long your script took
- use *http_load* to test a specific script as if a lot of web requests were done at the same time, and analyse the response time
- use *xdebug* to systematically track specific scripts and report a lot of information (memory used, execution time, overhead, calls to functions, etc)

We recommend using these analysis tests progressively and only get to the last method when you need a **considerable** improvement, as the technique itself is quite difficult to setup. The last technique is only considered in the *Dokeos Administration Level III* Training, as the application itself needs at least one day to setup and start using it.

Manual testing

This is very obvious, just try the *look and feel* approach and listen to your users. These are two generally very efficient methods to detect a slow process.

In-script checkpoints

You might have noticed that in some other web applications. Sometimes, the time the script took to execute is indicated at the bottom of the page, in the footer.

You can do this as well, very easily. At the beginning of a script (very first line), add this

```
$debug_start_time = microtime();
```

At the end of the script, add this

```
$debug_stop_time = microtime();

echo ($debug_stop_time - $debug_start_time).' ms';
```

This will display, at the bottom of the page, the number of microseconds that the script took to execute. You can then try several techniques to improve the current script and see if that affects the time to execute.

However, this technique is not perfect, as some performance issues might only appear when a lot of users are using the script at the same time. When you try it out yourself, you are alone so it doesn't reflect the same situation.

http_load

http_load is a script available on the net, that will let you check the performance of one of your scripts by simulating many simultaneous requests on your server.

You can get the script from here:

http://www.acme.com/software/http_load/

The script is used like this:

```
$ ./http_load -parallel 200 -seconds 10
http://my.dokeos.com/
```

This will try the URL http://my.dokeos.com/ with 200 simultaneous requests for 10 seconds.

The results look like this

```
49 fetches, 2 max parallel, 289884 bytes, in 10.0148
seconds

5916 mean bytes/connection

4.89274 fetches/sec, 28945.5 bytes/sec

msecs/connect: 28.8932 mean, 44.243 max, 24.488 min

msecs/first-response: 63.5362 mean, 81.624 max, 57.803
min

HTTP response codes:

  code 200 -- 49
```

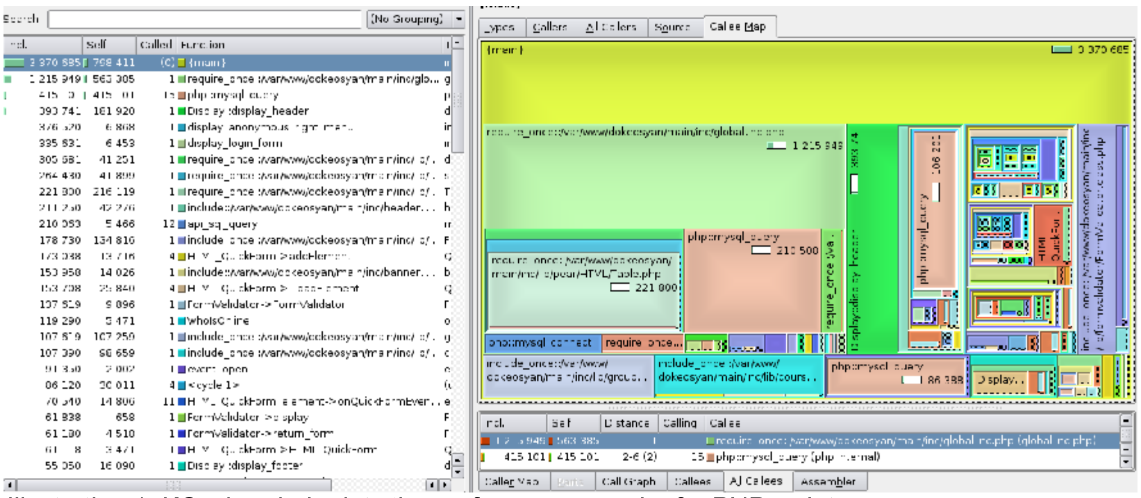
Basically, what is interesting here is the number of msecs for the connection and for the first response. Of course, if possible, you would like to make changes that will improve these.

Using Xdebug

Xdebug is a PHP extension that allows for very precise PHP tracing and debugging. You can find more information on the Xdebug website: http://www.xdebug.org/

However, the documentation is very technical and we only recommend to use it when absolutely necessary (very specific problem of slowliness that cannot be analysed in any other way).

This would generally allow you to get some detailed diagram of the script process that look like this (using kcachegrind)



Search: (No Grouping)

Incl.	Self	Called	Function
3 370 685	798 411	(C)	{main}
1 215 949	563 305	1	require_once::/var/www/dokeosyan/main/inc/glo...
415 [illegible]	415 [illegible]	15	php::mysql_query
393 741	181 920	1	Display::display_header
376 370	6 868	1	display_anonymous_right_menu
335 831	6 453	1	display_login_form
305 681	41 251	1	require_once::/var/www/dokeosyan/main/inc/...
264 430	41 899	1	require_once::/var/www/dokeosyan/main/inc/...
221 300	216 119	1	require_once::/var/www/dokeosyan/main/inc/...
211 250	42 276	1	include::/var/www/dokeosyan/main/inc/header...
210 093	5 466	12	api_sql_query
178 730	134 816	1	include_once::/var/www/dokeosyan/main/inc/...
173 038	13 716	4	HTML_QuickForm->addElement
153 958	14 026	1	include::/var/www/dokeosyan/main/inc/banner...
153 708	25 840	4	HTML_QuickForm->createElement
137 519	9 096	1	FormValidator->FormValidator
119 290	5 471	1	WhoIsOnline
107 879	107 250	1	include_once::/var/www/dokeosyan/main/inc/...
107 390	96 659	1	include_once::/var/www/dokeosyan/main/inc/...
91 330	2 002	1	event_open
86 120	20 011	4	<cycle 1>
70 510	14 806	11	HTML_QuickForm_element->onQuickFormEvent...
61 938	658	1	FormValidator->display
61 180	4 510	1	FormValidator->return_form
61 [illegible]	3 471	1	HTML_QuickForm->HTML_QuickForm
55 050	16 090	1	Display::display_footer

Types
Callers
All Callers
Source
Callee Map
{main}
3 370 685
require_once::/var/www/dokeosyan/main/inc/global.inc.php
1 215 949
php::mysql_query
215 500
require_once::/var/www/dokeosyan/main/inc/lib/pear/HTML/Table.php
221 000
php::mysql_connect
require_once...
include_once::/var/www/dokeosyan/main/inc/lib/group...
include_once::/var/www/dokeosyan/main/inc/lib/cours...
php::mysql_query
86 388
Display...
Caller Map
Parts
Call Graph
Callees
All Callees
Assembler

Incl.	Self	Distance	Calling	Callee
1 215 949	563 385	1		require_once::/var/www/dokeosyan/main/inc/global.inc.php (global.inc.php)
415 101	415 101	2-6 (2)	15	php::mysql_query (php::internal)

Illustration 1: KCachegrind prints the performance graph of a PHP script

Understanding and improving PHP performance

Caching

We already said PHP is an interpreted language. This is quite not the whole truth. Like Java, Perl and Python, it uses a so-called *bytecode* (called *opcode* in PHP's terminology) runtime. Unlike Java, but like the two others, the compilation is done each time you try to run the script.

Let us dig into this more deeply: source code (i.e. what you wrote) is read from disk and compiled in memory into a special binary form, architecture-independent, called *bytecode*. This *bytecode* is then executed by a special "emulator", also included in PHP.

This basically means that each time a user requires to view a PHP script, the interpreter (embedded in the web server when using the Apache PHP module), will read the script, compile it, execute it (getting data from the DB, etc...) and print the results to the user, usually in HTML, as we are talking about web scripting. This also means that if 10.000 users try to load the script at the same time, the web server will dumbly just read, execute and print the script 10.000 times, one after the other.

Hopefully, there are **techniques that allow the web server to reuse a previous bytecode** so that it only needs to execute and print, or techniques to cache the results. These techniques are called *caching* because you use the memory as a cache for a specific version of the script, that will come out and print when you need it. Whenever the script on the disk would be changed, the cache would be rebuild (ie the script compiled and its compiled form saved).

Caching can be done in several ways, from manual to automatic caching. However, manual caching is generally not really called *caching*, but rather *code optimization* in general. What we are going to see is caching done by specific extensions of the PHP language.

You can see a list of such PHP caching extensions available in Ubuntu by issuing the command:

```
$ apt-cache search php cache
```

This will give you a list containing the following packages:

```
php-cache - framework for caching of arbitrary data
php-cache-lite - Fast and lite data cache system
php5-memcache - memcache extension module for PHP5
php5-xcache - Fast, stable PHP opcode cacher
```

The extension we are going to analyse here is called *APC (Alternative PHP Cache)* and can be configured through the PHP configuration files. It is not in the list above because it is not shipped as a Debian/Ubuntu package (but is nonetheless easy to install).

The documentation for that PHP extension can be found here: http://www.php.net/apc

To install it, follow these steps

```
$ sudo pecl install apc
$ sudo vi /etc/php5/apache2/php.ini
```

Then add the following minimal APC extension section

```
extension=apc.so
[apc]
apc.enabled=1
```

And then reload your Apache server's configuration

```
$ /etc/init.d/apache2 reload
```

This should normally have installed the APC extension. Now you have a certain amount of options you can configure by adding them to the **php.ini** configuration file (in the [apc] section), following the descriptions given for each variable at http://www.php.net/apc

Persistent connections to DB

Maybe this will look suprising, but because of the nature of the web, the first recommended way to handle database connections was the single-request mode, which is applied by **opening a connection to a database once per script**, and close it directly afterwards.

This is because most websites are only just websites, and do not handle a lot of users data. You just come, consult two or three things, and go away.

When studying web applications, however, this can be a bit different. A user will connect to the website, login, use dozens of pages, change data, consult data, delete data, and will finally logout (this also explains why Dokeos does not benefit much from caches like *memcached*).

Because the model of web applications is different from the model of simple web sites, the single connection/disconnection per script can be a huge overhead.

There is a mechanism in PHP that allows for another mode. This other mode is called *persistent connection*. **In a persistent connection, the connection to the database is opened the first time** you enter the web application, and is closed when you logout. This means that for every single script you use, the connection will stay available, so you cut something like 1/10 second by script you load. This can quickly lead to a massive gain of performances, especially with many concurrent users.

But what are the drawbacks? Well, not many.

First, you will use **more memory**. Apache will need more, and MySQL will need more. This is probably not a problem considering the huge amount of RAM available in servers nowadays.

Second, you will **possibly** need to update the **number of opened connections allowed** at the same time **by the database server**.

Finally, you will need to **change the connection script** so that it uses the *persistent* mode rather than the normal one.

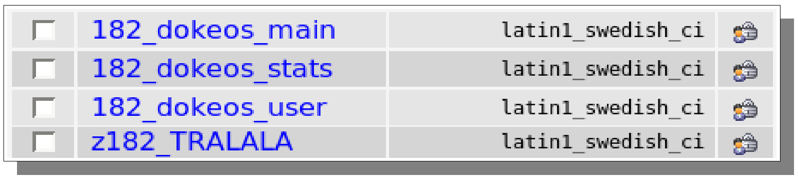
To change this in the Dokeos code, update the **main/inc/global.inc.php** file. Look for the call to *mysql_connect* and change it to *mysql_pconnect*.

This test should be accompanied by appropriate server load analysis, so that you can see if it makes a difference or not.

Database

Database structure

The first step in understanding how to improve the database's performance is understanding how it works and what are its weaknesses.



☐	182_dokeos_main	latin1_swedish_ci	
☐	182_dokeos_stats	latin1_swedish_ci	
☐	182_dokeos_user	latin1_swedish_ci	
☐	z182_TRALALA	latin1_swedish_ci	

Overall structure (1DB/course)

There are two modes that you can use to install Dokeos. The first mode is called *single DB* and was made only in an attempt to allow free users to use Dokeos with a free hosting provider, which would only give them the right to use one and only one database.

This mode, although functional, is **not recommended** for Dokeos campuses of more than 10 courses. In fact, it was flawed in the initial design.

Initially, the database only contains the Dokeos system tables (around 15). But when creating a course, the **50-or so tables needed for a course are added to the existing number of tables**. This works the same way for each new course, which leads you quite quickly to a number of 500 tables in the same database if you have 10 courses (even empty courses).

So what's wrong about that? We'll see this a bit later, in *Resources bottlenecks, Number of databases.*

The second database mode available when you install Dokeos is called the *Multiple databases* mode. This is only making things a little better, as instead of adding 50 tables to **the same database** each time, you are adding them to **another** database. This spreads the load for the server. We will see why in the *Resources bottlenecks, Number of databases.*

Regardless of the type of installation, you will find the same global structure in the Dokeos database(s). There is one set of system tables (or *main* tables), one set of statistics tables, one set of users tables and an unlimited number of sets of courses tables. You can find all the following diagrams on the Dokeos' public wiki: http://www.dokeos.com/wiki/index.php/Database_schema_1.8

These diagrams are provided here for your comfort...

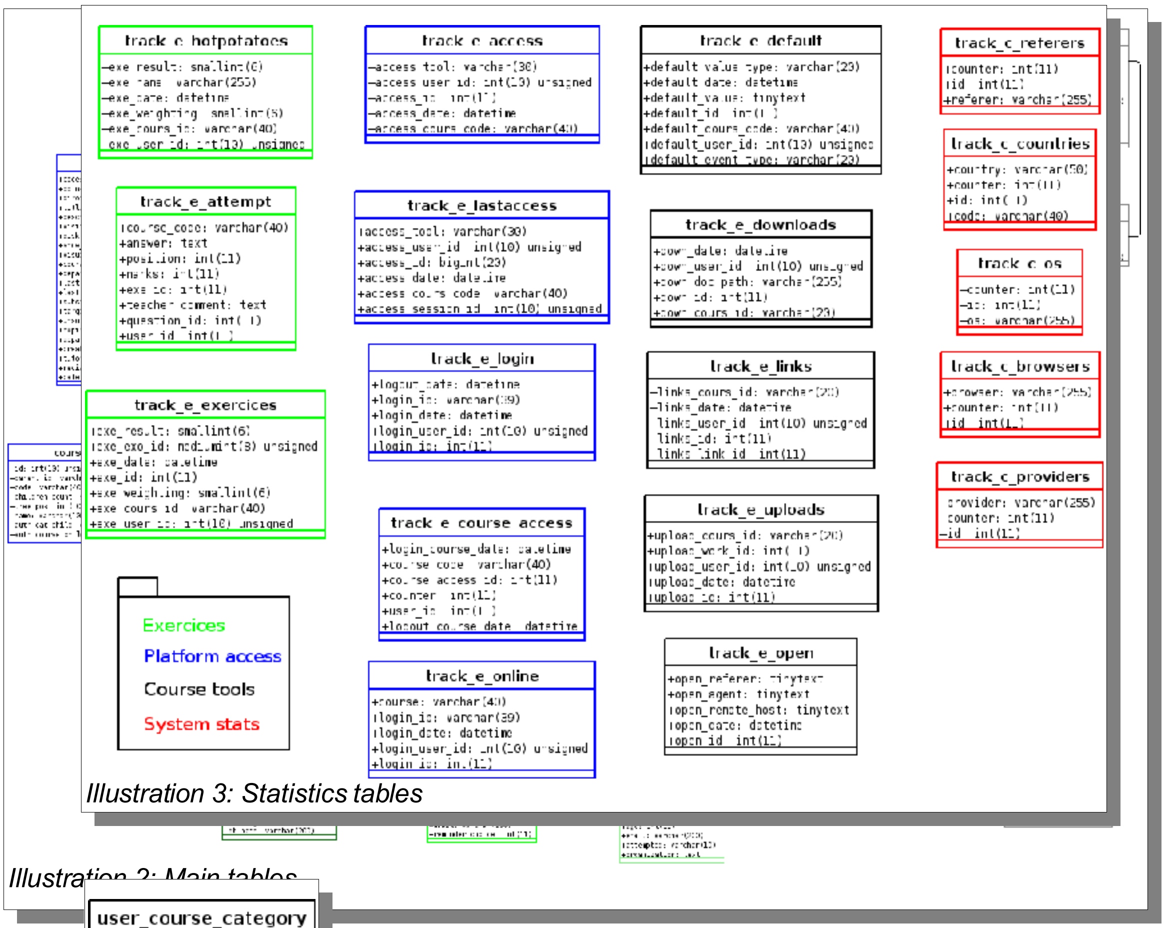


Illustration 3: Statistics tables

Illustration 2: Main tables

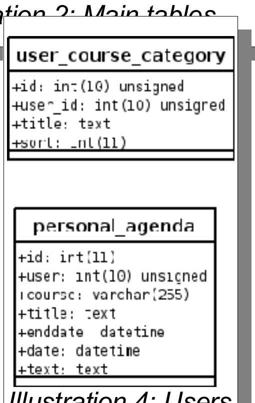


Illustration 4: Users tables

The users tables are a recent addition to the Dokeos database structure and contain data relative to the user. It can be argued that these two tables would have fit better in the main set of tables, but the problem was not really discussed internally before the addition was made.

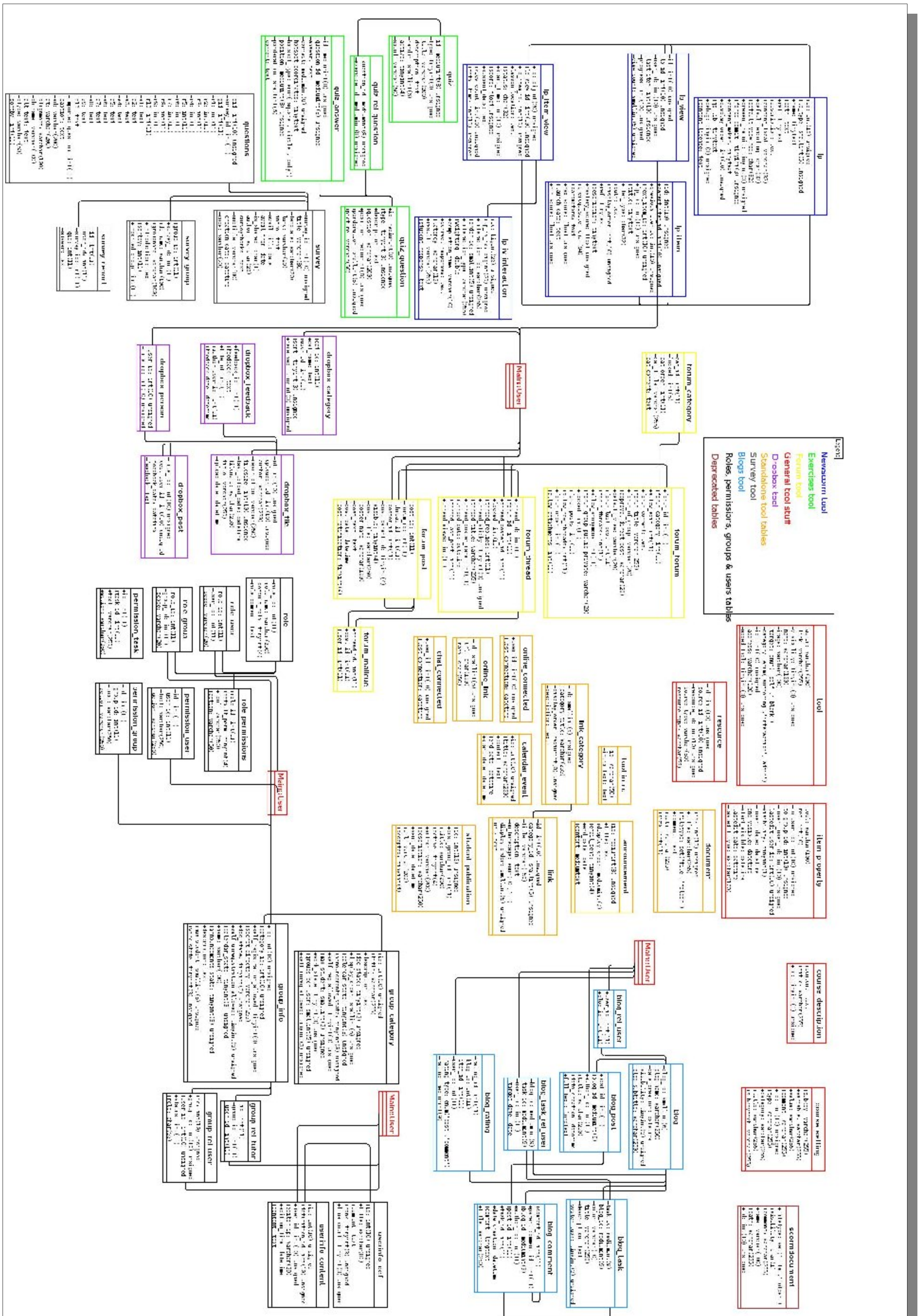


Illustration 5: Course tables

As you can see on those diagrams (from the database structure of Dokeos 1.8.0), there are lots of tables. As you can imagine, repeating this structure too many times might make the whole system even more complex. This is why, currently, **the recommended database structure is to have one database per course**.

Main databases

The main databases (plural) can be considered as being the databases and tables that do **not** get duplicated when a course is created. As such, the system/main tables (courses, users, sessions, settings, ...), the statistics tables (course accesses, exercises results, ...) and the users tables (personal agenda, personal courses categories) can be considered as part of the main databases.

Let's analyse them a bit further... (taking a look at the diagrams)

The main database contains:

– a list of **courses** and their relations to users, sessions, etc
– a list of **users** and their relations to courses, sessions, surveys, etc
– a list of **sessions** and their relations to courses and users
– a list of **course categories** (to group the courses)
– a list of **surveys** and their relations...
– a list of **settings** and their corresponding options

These informations are pretty basic and they will be used each time a user logs in, each time a user enters a course, each time he enters the session, and each time a new page is loaded (as the global settings need to be reloaded to avoid memory effect that could prevent a change from having effect on the users already connected). There are a **lot of read** operations but not many **write** operations.

The statistics database contains very dynamic data. In these tables, several lines are written for each page loaded. There are more **write** operation than **read** operations.

The database contains:

– login/logout and connection time
– exercises results
– users' origins (where they came from before coming on the portal, etc)
– some users' actions

The users database is irrelevant here, so we will just pass it.

Courses databases

The courses databases (plural when there are more than one course) contain all the information related to a course:

- forums
- documents
- exercises (but not results, which are stored in the stats database)
- learning paths (as well as results)
- links
- course settings
- users document sharing
- etc.

The information inside the courses databases is generally **read** a lot but not **written** a lot, although generally written more often than the main database.

Analysing MySQL performance

Number of databases

We talked about the number of databases in the previous section (*Database structure*) and said we would leave the analysis of the problem caused by many tables for later. Well, here we are.

Before we start with an example, it is important to know that MySQL stores its databases as directories, and its tables as files inside those directories.

1 database = 1 directory

1 table = 3 files (content, structure and index)

If 1 database contains 50 tables, the corresponding directory will contain 150 files.

Now let's go back to the analysis of the *Single database* mode. What we saw was that for each course, 50 new tables we added to the same database. That was giving us a large 500 tables in one database for an installation with 10 courses.

Now 500 tables mean that we have 1500 files (500*3) in the database's directory.

One thing that is not easily understood is that MySQL relies on the file system (Fat32, Ext2, Ext3, ReiserFS, NTFS, ...) to store the files of the database. When there are 1500 files in the database's directory, it is up to the file system to handle the accesses to the files in a quick way.

Now to give you an idea, anything going above 1000 entries in one single directory is probably not a good thing. Some file systems will simply slow down a little, others will slow down dramatically, and others will even stop accepting new files after a certain limit (around 50.000 for example).

What we have to understand is that **any** solution to reduce the number of files in one single directory is a good solution. This is why the *multiple databases* install type is better. Because rather than putting 500 tables in one database (so 1500 files in one directory), we create several databases (=directories) that, each, holds no more than 50-and-something tables (150 files).

The ideal solution, however, would be to redesign the database so that it uses a fixed number of tables and databases (one database and a fixed number of tables) for all the courses. This is what we are trying to achieve in the next major version of Dokeos, Dokeos LCMS.

In the meantime, try to remember that you have to spread the load and try to always have a reasonable amount of tables in one database.

Slow Queries

Slow queries are just what their name says. Queries that are slow.

MySQL offers a way to identify those slow queries by logging them into a file on the system. This mode can be activated in the MySQL configuration file: */etc/mysql/my.cnf*

```
#log_slow_queries       = /var/log/mysql/mysql-slow.log
#long_query_time = 2
```

Activating this option (removing the #) will start recording all the slow queries. The *long_query_time* option is the number of seconds after which a query is considered ***slow***.

Note that 2 seconds on a web server is already very slow, because if many users access the website simultaneously, the site will slow down while waiting for the query results.

The results of the slow queries reports are very easy to understand. They give you the full query that was executed as well as the time taken to execute.

This alone cannot improve your system's performances, but only allows you to analyse the current situation to apply the following techniques.

Indexes

Indexes are a way to improve the performance of all the queries on a table, by having another set of data, in parallel to the existing table. The algorithms are generally quite complex but we will not see them in detail here.

What we are interested in is what we can do with them.

An index is a way to get faster to the data, using “shortcuts”. For example, let's imagine a table with 1 million rows. Each of these rows has a unique identifier and a title (string of characters). Some of these titles are the same, or start in the same way.

Now when we search the contents of that table, we will generally search for a title, not the ID. In a normal case (without index), we would have to go through the whole table, row by row, for 1 million rows. This would take a lot of time.

An index is a kind of shortcut table, where every title can be found more quickly, and for each title, there is a list of corresponding IDs in main the table.

This means that we will find every title much more quickly, and also that we use a little more space to create this shortcuts table.

When finding a slow query in the **slow queries** logs as described previously, the first step to take is to check that there are enough indexes in the fields used by the query.

For example, let's analyse this slow query:

```
# Time: 070630 20:26:20
# User@Host: cepec[cepec] @ localhost []
# Query_time: 5  Lock_time: 0  Rows_sent: 59
Rows_examined: 9395035
SELECT count(*) AS number_of_posts, posts.forum_id FROM
`dokeos_CAERMATH`.`forum_post` posts,
`dokeos_CAERMATH`.`forum_thread` threads,
`dokeos_CAERMATH`.`item_property` item_properties
     WHERE posts.visible=1
     AND posts.thread_id=threads.thread_id
     AND threads.thread_id=item_properties.ref
     AND item_properties.visibility=1
     AND item_properties.tool='forum_thread'
     GROUP BY threads.forum_id;
```

This bit has been taken from one of our current slow-queries logs.

What can we find here?

First, let's pick out the tables that we use:

forum_post, *forum_thread* and *item_property* of the course database *dokeos_CAERMATH*.

As the name shows, we are looking at the tables for the forum tool. What do we notice in the query? We are querying *thread_id, visible, tool* and *ref.* Generally speaking, a column that only contains 1 or 0 (like the *visible* column) will not gain a lot from indexing.

The others though, like *item_properties.tool* and *item_properties.ref* will potentially gain a lot, because they are strings and because strings are difficult to handle by a computer (compared to integers).

On the other hand, the largely-used *thread_id* is a unique ID as the Dokeos nomenclature shows (ends with *_id*), so it will most likely be indexed already.

So, using *phpMyAdmin*, we are going to check, for each table concerned, if they have the indexes we expect or not. This can be done by looking, in the definition view of each table, what is shown in the *Indexes* box.

Adding an index is very easily done by clicking, next to the column's definition, on the small icon with a lightning bolt.

SQL Joins

Joins in SQL are a way to link two (or more) tables together. By joining the *users* tables and the *courses* table, you could print out a list of users and, for each user, the titles of the courses he is subscribed to.

This could not be done directly in the database if you did not use a join.

Joins can be done in two distinct ways. You can use a query with a long *SELECT* clause like we saw for the slow queries, or you can use the *JOIN* syntax. There is no real difference except that the *JOIN* will let you do things a bit more complicated (like only get elements if they are not null on both sides of the join).

Anyway, in the context of this training, we only want one thing regarding the JOINs: optimization.

Well, optimization of the JOIN queries is actually made the same way than in the previous section on indexes. You add indexes on the fields that are used in the join, and this improves the speed of the query.

Sometimes, you can also improve the speed dramatically by adding duplicates in your table. For example, for the course titles given above, we could change the PHP code so that each time a user is subscribed to a course, we add the course in a new field called "courses_titles" in the user table. This way, we will not need to query the *course* table when we want to get a list of users and the courses they are subscribed to.

This technique, however, can only be applied by modifying the application code (Dokeos PHP code in this case) and should not be applied lightly. This also has the side effect of making your database use more space.

Caching

Caching, as we have seen it before, is a technique to use more the memory of the system to avoid disk accesses, which are slow.

Caching in MySQL can be done by changing the MySQL server settings in */etc/mysql/my.cnf*:

```
#
# * Fine Tuning
#
key_buffer              = 16M
max_allowed_packet      = 16M
thread_stack            = 128K
thread_cache_size       = 8
#max_connections        = 100
#table_cache            = 64
```

```
#thread_concurrency     = 10
#
# * Query Cache Configuration
#
query_cache_limit       = 1M
query_cache_size        = 16M
```

These settings can be updated to:

- increase the number of indexes kept in memory (16M here)
- increase the memory size given to each thread (8M)
- increase the maximum number of connections (100)
- increase the number of tables that can be kept simultaneously in cache (here 64)
- update the limit of memory per query and the total memory for queries

Filesystem performances

It is important to note that some OS-bound values are important for performances.

We already covered the importance of the underlying filesystem for MySQL databases. But Dokeos also uses many files accessed directly through PHP on the filesystem somewhere. User uploaded files, PHP sessions, temporary files, pictures, ... are all searched and read or written from or to the filesystem.

Incidently, taking care of the filesystem performance can have drastic effect on Dokeos.

Running Dokeos for a certain time may show some files are never cleaned, though won't never be used anymore. Day after day, the amount (we already noted some reaching millions of files) of those files in important, often-accessed, directories are likely to slow down Dokeos (and also its maintenance).

Notably, the *dokeos/main/garbage/* and *dokeos/sessions/* are good candidates.

Solutions are:

1) to make sure the filesystem we use (Ext2, Ext3, ReiserFS, ...) is adequately tuned to access directories with many files. On Ext3, the *dir_index* parameter is crucial.

To check it has been set for the filesystem hosting Dokeos files, use the following command:

```
dumpe2fs /dev/sda1|grep dir_index
```

If the output is empty, it is not set, and you should set it like this:

```
tune2fs -O dir_index /dev/sda1
```

The directories created from now on will behave far better if storing many files.

2) to periodically clean those directories. Writing scripts and make them run automatically at schedule.

Monitoring system performances on Linux

Performances snapshot

Using standard tools

ps

top

sar

Performances over time

A proposed tool: Munin

Automating processes

In the common day-to-day job of a Dokeos administrator, you will find there are many ways in which you can make things more automatic. For example, you could automate the cleaning of the garbage directories, or ensure the permissions are always right when uploading files through Dokeos, or prepare a script that sends you an e-mail when the database server is down.

All this can be done through Shell scripts, scripts that will run on the server and execute system commands periodically (using Cron jobs).

In this section, as an example, we will see how to build a script that cleans the main/garbage directory every week.

Writing the script

The script is easily written. You know you want to clean your *dokeos/main/garbage/* directory contents. You know as well where it is located on your system (*/home/dokeos/main/garbage/*), and you know how to delete things on the command-line: *rm* *

Now how would you put this all together in a script? Very easily still. Open a blank text file and call it “clean_dokeos_garbage.sh”

Edit the file and add

```
rm /home/dokeos/main/garbage/*
```

Your work is done here. If you exit the script and try to run it, it will execute the requested command. To do that, you need something like

```
sh clean_dokeos_garbage.sh
```

Setting up the cronjob

Now that you just wrote your script, you need to find a way to execute it every week. Let's say that you want to do that every Sunday at 11pm.

The cronjob system is a very convenient way to operate automatic commands. You just set the time and date you want it to run and off you go.

Usually, you would start the following command as the user which needs to execute the job to edit its cronjob list:

```
cronjob -e
```

On a Debian/Ubuntu system (and possibly other UNIX/Linux ones), you can just create a file in some of */etc/cron.daily*, */etc/cron.weekly*, */etc/cron.monthly* or */etc/cron.d* directories (actually, format may vary slightly).

The format of a line there would then be something like:

```
*/15 * * * * www-data /usr/local/bin/myscript.sh
```

And it would mean:

Each 15 minutes (*/15) of every hour, of every day, of every month, whatever day of week (ex.: Sunday), execute as user “www-data” the command */usr/local/bin/myscript.sh*.

Wildcards (*), ranges (ex.: 8-11) and lists (ex.: 8,11) are allowed in schedules.

Finding and reporting bugs

Logging bugs

php.ini -> display_errors = Off
    -> log_errors = On => logged into /var/log/apache2/host-errors.log

Dokeos test mode can be activated from the administration area and will display the SQL queries in full when an error occurs but you cannot run in debug mode for only one user (the whole campus has to be switched).

Reporting bugs

The best reports always follow a certain set of rules:

- was reported in http://projects.dokeos.com in the appropriate category (and has its own category if special support contract)
- ships with an meaningful screenshot of the error, including the URL
- gives the URL (even if it is already in the screenshot)
- gives the login used (but not the password)
- indicates the Dokeos version
- indicates a way to reproduce the bug (succession of actions to get to the error)
- uses the severity with respect (it is not because it's your portal that it has to be *high*. A *high* severity is something that blocks your system. The priority can be set at taste, depending on the real priority of the task.

Conclusion

In this training, we have seen

- how to prepare a server for a Dokeos campus
- how the code of Dokeos was structured
- how to improve the Dokeos code performance
- how the Dokeos databases were structured
- how the database performance could be improved
- how to automate administration processes

In general terms, the system part of improving performances (caching, database indexing, filesystem) should be applied first, as this will not affect the upgrades to the next version of Dokeos. Then, if performance is really an issue, and if code changes is the only solution left, these changes should be reported back to the Dokeos development team as soon as possible so they can be integrated, avoiding the extra cost of re-applying the changes on next upgrade.