PHP Linux Daemon erstellen

== Beispiel ==
Unser Daemon hört im Beispiel auf den Namen "veryimportant", was übersetzt aus dem Englischen für "sehr wichtig" steht.
Nach gültigen Konventionen wird der Daemon gestartet über ein Init Script.

Quellcode

```
1. /etc/init.d/veryimportant start
```

== Linux Standard Base (LSB) ==
Die Linux Standard Base definiert eine Binärschnittstelle mit dem Ziel, die Kompatibilität zwischen den verschiedenen Linux-Distributionen zu verbessern.
Das Script unter /etc/init.d/veryimportant sieht wie folgt aus:

Quellcode

```
1. #!/bin/sh
2. ### BEGIN INIT INFO
3. # Provides: php-queue
4. # Required-Start: $remote_fs mysql
5. # Required-Stop: $remote_fs mysql
6. # Should-Start: $network $named $time
7. # Should-Stop: $network $named $time
8. # Default-Start: 2 3 4 5
9. # Default-Stop: 0 1 6
10. # Short-Description: Starts and stops the veryimportant php daemon.
11. # Description: Controls the veryimportant PHP script for me, so that
12. # processing is active in all run-levels in which network
13. # services are active
14. ### END INIT INFO
15. #
16. # Author: Max Mustermann <max@mustermann.de>
17. #
18. # This init script conforms to LSB 3.2.0.
19. #
20. PATH=/bin:/usr/bin:/sbin:/usr/sbin
21. DAEMON=/home/veryimportant/daemon.php
22. PIDFILE=/var/run/veryimportant.pid
23. test -x $DAEMON || exit 0
24. . /lib/lsb/init-functions
25. case "$1" in
26. start)
27. log_daemon_msg "Starting veryimportant processing" "veryimportant.php"
28. start_daemon -p $PIDFILE $DAEMON
29. log_end_msg $?
30. ;;
31. stop)
32. log_daemon_msg "Stopping veryimportant processing" "veryimportant.php"
33. killproc -p $PIDFILE $DAEMON
34. log_end_msg $?
35. ;;
36. force-reload|restart)
37. $0 stop
38. $0 start
39. ;;
```

```
44. *)
45. echo "Usage: $0 {start|stop|restart|force-reload}" >&2
46. exit 2
47. ;;
48. esac
```

Alles anzeigen

Wir machen das Script nur für root ausführbar, aber für alle lesbar.

Quellcode

```
1. chmod 0755 /etc/init.d/veryimportant
```

== Das PHP Script ==
Das Script des Hauptprogramms legen wir unter /home/veryimportant/daemon.php ab.
Die Kommandozeilenparameter "d" und "u" sind aktiviert. Mit "d" schalten wir in den Debug Modus und mit "u" nehmen wir unterschiedliche Benutzer an. Das Script soll nicht unter root, sondern einem anderen Benutzer laufen.
Bei uns "veryimportant".

Wir legen den Benutzer an mit

Quellcode

```
1. adduser veryimportant
```

Das Script enthält einen Logger, der entsprechende Ausgaben an einen Output Stream leitet. Das kann einer Logdatei oder der Standard-Output sein.
Die Log Datei befindet sich unter /var/log/veryimportant.log.
Damit der Daemon nicht mehrfach gestartet werden kann verwenden wir eine PID Datei unter /var/run/veryimportant.pid.

Nun zum Quellcode:

Quellcode

```
1. #!/usr/bin/php
2. <?php
3. $options = getopt('du:');
4. if ($options === FALSE) {
5. die("Failed to parse command line options\n");
6. }
7. $debug = isset($options['d']);
8. $user = isset($options['u']) ? $options['u'] : "veryimportantdaemon";
9. $pidfile = "/var/run/veryimportant.pid";
10. if (getmyuid() != 0) {
11. die("This script must be started as root (it will revoke its privileges)\n");
12. }
13. if (!($passwd = posix_getpwnam($user))) {
14. die("Failed to get passwd entry for user \"$user\"\n");
15. }
16. # Set the default timezone to the system's timezone.
17. if (function_exists("date_default_timezone_set") and
18. function_exists("date_default_timezone_get"))
19. @date_default_timezone_set(@date_default_timezone_get());
20. if (file_exists($pidfile)) {
21. $pid = rtrim(file_get_contents($pidfile));
22. if (posix_kill($pid, 0)) {
```

```
30. die("Failed to create pidfile $pidfile (PID $pid is still running)\n");
31. }
32. trigger_error("Removing stale pidfile $pidfile", E_USER_NOTICE);
34. unlink($pidfile);
35. }
36. # if debug ist not set open a child process
38. if (!$debug) {
39. $pid = pcntl_fork();
40. if ($pid == -1) {
41. die("Could not fork");
42. } else if ($pid) {
43. // We are the parent, we may exit now.
44. exit(0);
45. }
46. }
48. $pid = pcntl_fork();
49. if ($pid == -1) {
50. die("Could not fork again");
51. } else if ($pid) {
52. pcntl_signal(SIGINT, SIG_IGN);
53. pcntl_signal(SIGTERM, SIG_IGN);
55. // We are the parent. Write pidfile and wait until the child exists.
56. if (!file_put_contents($pidfile, "$pid\n")) {
57. posix_kill($pid, 15);
58. die("Failed to create pidfile $pidfile");
59. }
61. $child = pcntl_waitpid($pid, $status);
62. unlink($pidfile);
63. exit;
64. }
66. $uid = $passwd['uid'];
67. $gid = $passwd['gid'];
69. if (!posix_setgid($gid) or !posix_setuid($uid)) {
70. die("Failed revoke privileges to run as user \"$user\"\n");
71. }
73. function sig_handler($signo)
74. {
75. switch ($signo) {
76. case SIGINT:
77. case SIGTERM:
78. // handle shutdown tasks
79. Logger::info("Got signal $signo, exiting");
80. exit(128 + $signo);
81. }
82. }
84. function err_handler($errno, $errstr, $errfile, $errline, $errcontext)
85. {
86. switch ($errno) {
87. case E_USER_ERROR:
88. // print and exit
89. Logger::error("Fatal error: $errstr in $errfile on line $errline (exiting)");
91. case E_USER_WARNING:
92. Logger::warn("Warning: $errstr in $errfile on line $errline");
93. break;
94. case E_USER_NOTICE:
95. Logger::info("Notice: $errstr in $errfile on line $errline");
96. break;
97. default:
```

```
98.  Logger::warn("Unknown error type $errno: $errstr in $errfile on line $errline");
99.  break;
100. }
101. return TRUE;
102. }
103. /**
104. * basic logger
105. *
106. * Example usage:
107. * @code
108. * Logger::init(fopen("/var/log/veryimportant.log", "a"));
109. *
110. * Logger::info("Small info line");
111. * Logger::warn("A little warning");
112. * Logger::debug("Some sweet debug information");
113. * Logger::error("Houston, we've got a (serious) problem");
114. * @endcode
115. */
116. class Logger {
117. private static $instance = NULL;
118. private $handle = null;
119. protected function __construct($handle) {
120. $this->handle = $handle;
121. register_shutdown_function(array(&$this, "__destruct"));
122. }
123. public function __destruct() {
124. if($this->handle) {
125. fclose($this->handle);
126. }
127. return true;
128. }
129. public function writeLog($level, $message) {
130. if($this->handle) {
131. fwrite($this->handle, sprintf("%s - %s - %s\n", date('r'), $level, $message));
132. }
133. }
134. private final function __clone() {}
135. private static function getInstance() {
136. if (self::$instance === NULL) {
137. throw new Exception('call init before');
138. }
139. return self::$instance;
140. }
141. public static function init($output) {
142. self::$instance = new self($output);
143. }
144. public static function info($message) {
145. self::getInstance()->writeLog('INFO', $message);
146. }
147. public static function warn($message) {
148. self::getInstance()->writeLog('WARN', $message);
149. }
150. public static function debug($message) {
151. self::getInstance()->writeLog('DEBUG', $message);
152. }
153. public static function error($message) {
154. self::getInstance()->writeLog('ERROR', $message);
155. exit;
```

```
168.  }
169.  }
170.  set_error_handler("err_handler");
171.  pcntl_signal(SIGINT, "sig_handler");
172.  pcntl_signal(SIGTERM, "sig_handler");
173.  # Enable the signal handle callback mechanism.
174.  declare(ticks = 1);
175.  # Open the output stream which should receive all log messages.
176.  # if debug is set just print the messages - if not, then log to file
177.  Logger::init($debug ? STDOUT : fopen("/var/log/veryimportant.log", "a"));
178.  # program code
179.  # run controller from here
180.  # e.g. Controller::dispatch()
181.  while(true) {
182.  Logger::info("i am running");
183.  sleep(1);
184.  }
185.  ?>
```

Alles anzeigen