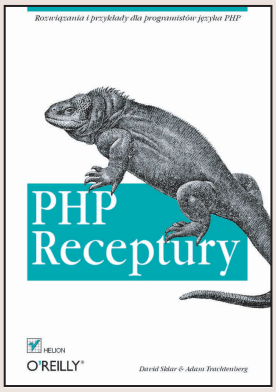
PHP. Receptury



Autorzy: David Sklar, Adam Trachtenberg
Tłumaczenie: Tomasz Jarzębowicz
ISBN: 83-7361-117-7
Tytuł oryginału: PHP Cookbook
Format: B5, stron: 648

PHP to prosty ale jednocześnie bardzo użyteczny język skryptowy dostępny na zasadach open source. W ostatnich latach stał się jednym z ważniejszych narzędzi programistycznych w Internecie. Ponad milion witryn, od stron wielkich korporacji po strony prywatne, wykorzystuje PHP do dynamicznego generowania zawartości stron WWW. PHP zawiera zbiór bardzo przydatnych funkcji, ma prostą składnię, obsługuje wielu różnych systemów operacyjnych oraz usług sieciowych, co czyni go idealnym narzędziem do szybkiego tworzenia dynamicznych stron WWW.

Książka zawiera unikalny zbiór przykładów i rozwiązań problemów spotykanych na co dzień w trakcie programowania w języku PHP. Dla każdego problemu zostało przedstawione wyczerpujące rozwiązanie lub „receptura" – krótki fragment kodu, który od zaraz można zastosować w aplikacjach. Jednak ta książka to nie tylko gotowy do wykorzystania kod, znajdziesz tu także wyjaśnienie, jak i dlaczego przedstawiony kod działa, a to ułatwi dostosowanie go do rozwiązania podobnych, a nie tylko identycznych problemów.

Receptury z niniejszej książki rozwiązują nie tylko proste problemy, jak wysyłanie zapytania do bazy danych lub przetwarzanie adresów URL, ale także całe programy wykonujące złożone zadania, na przykład wyświetlanie tabel HTML lub generowanie wykresów słupkowych.

Książka zawiera ponad 250 receptur obejmujących między innymi:

- pracę z prostymi typami danych: tekstami, liczbami, datami, czasem i tablicami,
- bloki tworzenia programów w PHP: zmienne, funkcje, klasy i obiekty,
- programowanie sieciowe, czyli tworzenie formularzy, dostęp do baz danych i XML,
- użyteczne funkcje jak: wyrażenia regularne, szyfrowanie i bezpieczeństwo, grafika, internacjonalizacja i lokalizacja a także usługi sieciowe,
- pracę z plikami i katalogami,
- wiersz poleceń PHP i PHP-GTK,
- PEAR (PHP Extension and Application Repository)

Książka zawiera wiele kodu użytecznego dla wszystkich programistów PHP, od nowicjuszy po weteranów. Zamiast śledzić listy dyskusyjne, dokumentację dostępną w Internecie oraz inne źródła, wystarczy zajrzeć do „PHP. Receptury", by znaleźć rozwiązania typowych problemów.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

5

Zmienne

5.0. Wprowadzenie

Zgodnie z logiką warunkową zmienne są tym, co czyni programy komputerowe silnymi i elastycznymi. Jeżeli potraktujemy zmienną jako opatrzone nazwą pudełko, które przechowuje jakąś wartość, możemy powiedzieć, że PHP pozwala na tworzenie dziurawych, pustych pudełek, pudełek, które zawierają nazwy innych pudełek, pudełek wypełnionych liczbami lub łańcuchami znaków, pudełek przechowujących tablice innych pudełek, pudełek wypełnionych obiektami oraz innych odmian pudełek, jakie tylko można sobie wyobrazić.

Zmienna może być zarówno ustawiona, jak i nieustawiona. Zmienna z przypisaną do niej jakąkolwiek wartością: `true` lub `false`, pustą lub niepustą, jest ustawiona. Funkcja `isset()` zwraca wartość `true`, jeżeli przekazano do niej zmienną, która jest ustawiona. Jedynym sposobem, aby zmienną ustawioną przekształcić w zmienną nieustawioną, jest wywołanie wobec niej funkcji `unset()`. Do `unset()` można przekazywać skalary, tablice i obiekty. Do funkcji tej można również przekazać jednocześnie wiele zmiennych, aby wszystkie uczynić nieustawionymi:

```
unset($vegetables);
unset($vegetables[12]);
unset($earth, $moon, $stars);
```

Jeśli zmienna jest obecna w łańcuchu zapytania URL, to będzie ona ustawiona nawet wówczas, jeśli nie przypisano do niej żadnej wartości. A zatem:

```
http://www.przyklad.com/set.php?szympansy=&malpy=12
```

spowoduje ustawienie `$_GET['malpy']` na `12`, a `$_GET['szympansy']` będzie pustym łańcuchem znaków.

Wszystkie nieustawione zmienne są również puste. Zmienne ustawione mogą być puste lub niepuste. Puste zmienne mają wartość logiczną równą `false`, są to: liczba całkowita 0, zmienna typu `double` 0.0, pusty łańcuch znaków, łańcuch znaków postaci „0", wartość logiczna `false`, tablica nie zawierająca żadnych elementów, obiekt bez zmiennych i metod oraz wartość `NULL`. Wszystkie inne zmienne są niepuste. Dotyczy to również łańcucha znaków „00" oraz łańcucha „ ", który zawiera tylko znak spacji.

Zmienne mogą być odczytywane jako mające wartość albo `true`, albo `false`. Powyżej wymieniono wszystkie wartości, które przez PHP traktowane są jako równoważne wartości logicznej `false`. Każda inna wartość jest wartością `true`. Różnica pomiędzy tym, co puste, a tym, co fałszywe polega na tym, że tylko zmienne mogą być puste. Stałe oraz wartości zwracane przez funkcję mogą mieć wartość `false`, ale nie mogą być puste. Na przykład poniższa instrukcja jest prawidłowa, ponieważ `$first_name` jest zmienną:

```
if (empty($first_name)) { .. }
```

Natomiast poniższe dwa przykłady spowodują wygenerowanie błędu parsera, ponieważ `0` (stała) oraz wartość zwracana przez funkcję `get_first_name()` nie mogą być puste:

```
if (empty(0)) { .. }
if (empty(get_first_name())) { .. }
```

5.1. Unikanie pomyłek między operatorami == i =

Problem

Podczas porównywania zmiennej i stałej chcemy uniknąć przypadkowego przypisania wartości.

Rozwiązanie

Napiszemy:

```
if (12 == $dwarves) { ... }
```

zamiast

```
if ($dwarves == 12) { ... }
```

Umieszczenie stałej po lewej stronie wywoła błąd parsera w razie zastosowania operatora przypisania. Innymi słowy, PHP wyświetli błąd, jeżeli napiszemy:

```
if (12 = $dwarves) { ... }
```

natomiast:

```
if ($dwarves = 12) { ... }
```

zostanie wykonane bez ostrzeżeń, czyli `12` zostanie przypisane do zmiennej `$dwarves`, a następnie uruchomiony zostanie kod wewnątrz bloku (ponieważ wynikiem operacji `$dwarves = 12` jest wartość `12`, traktowana jako `true`).

Analiza

Umieszczenie stałej po lewej stronie porównania wymusza przyrównanie do typu stałej. Powoduje to problemy w sytuacji, gdy liczba całkowita porównywana jest ze zmienną mogącą być liczbą całkowitą lub łańcuchem znaków. Jeśli `$dwarves` ma wartość `0`, wyrażenie `0 == $dwarves` ma wartość `true`. Jednak taką samą wartość będzie miało to wyrażenie w sytuacji, gdy `$dwarves` będzie mieć wartość `śpioch`. Jeżeli liczba całkowita (`0`) znajdzie się po lewej stronie operatora porównania, PHP przekształci przed wykonaniem operacji porównania to, co jest po prawej stronie (łańcuch znaków `śpioch`) na liczbę całkowitą (`0`). Aby tego uniknąć, należy zastosować operator tożsamości `0 === $dwarves`.

Zobacz również

Dokumentacja operatora = jest dostępna pod adresem *http://www.php.net/language.operators.assignment*, a dokumentacja operatorów == i === — na stronie *http://www.php.net/manual/language.operators.comparison.php*.

5.2. Ustalanie wartości domyślnej

Problem

Chcemy przypisać wartość domyślną do zmiennej, która jeszcze nie posiada żadnej wartości. Często zdarza się, że chcemy w sposób bezpośredni przypisać wartość domyślną zmiennej, która może zostać nadpisana danymi z formy wejściowej lub przez zmienną środowiskową.

Rozwiązanie

Należy zastosować funkcję `isset()` w celu przypisania wartości domyślnej zmiennej, która może już posiadać wartość:

```
if (! isset($cars)) { $cars = $default_cars; }
```

Aby nadać nowej zmiennej wartość (na przykład wartość domyślną), należy zastosować operator potrójny `(a ? b : c)`:

```
$cars = isset($_REQUEST['cars']) ? $_REQUEST['cars'] : $default_cars;
```

Analiza

Stosowanie funkcji `isset()` ma zasadnicze znaczenie podczas przypisywania wartości domyślnych. Bez niej wartość nie mająca charakteru domyślnego nie może być zerem lub jakąkolwiek inną wartością, obliczaną jako `false`. Rozważmy następujące przypisanie:

```
$cars = $_REQUEST['cars'] ? $_REQUEST['cars'] : $default_cars;
```

Jeżeli `$_REQUEST['cars']` ma wartość `0`, `$cars` jest ustawiona na `$default_cars` i to pomimo tego, że `0` może być prawidłową wartością zmiennej `$cars`.

Aby w łatwy sposób ustawić jednocześnie wiele wartości domyślnych, można zastosować tablicę z wartościami domyślnymi. Kluczami w tablicy wartości domyślnych są nazwy zmiennych, a wartości znajdujące się w tablicy są wartościami domyślnymi dla każdej zmiennej:

```
$defaults = array('imperatorzy' => array('Rudolf II','Kaligula'),
                  'warzywo'     => 'seler',
                  'hektary'     => 15);

foreach ($defaults as $k => $v) {
    if (! isset($GLOBALS[$k])) { $GLOBALS[$k] = $v; }
}
```

Ponieważ zmienne są ustawione w globalnej przestrzeni nazw, poprzedni kod nie zadziała, jeśli będziemy chcieli ustawić wartości domyślne obowiązujące jedynie wewnątrz funkcji. Aby wykonać tę operację, należy zastosować zmienne zmiennych:

```
foreach ($defaults as $k => $v) {
    if (! isset($$k)) { $$k = $v; }
}
```

Zobacz również

Dokumentacja dotycząca funkcji `isset()` jest dostępna pod adresem *http://www.php.net/isset*; zmienne zmiennych są omówione w recepturze 5.4 oraz pod adresem *http://www.php.net/language.variables.variable*.

5.3. Wymiana wartości bez używania zmiennych tymczasowych

Problem

Chcemy wymienić wartości w dwóch zmiennych bez użycia dodatkowych zmiennych do przechowywania tych wartości.

Rozwiązanie

Aby zamienić zmienne `$a` i `$b`, napiszemy:

```
list($a,$b) = array($b,$a);
```

Analiza

Konstrukcja językowa PHP o nazwie `list()` umożliwia przypisywanie wartości z tablicy do pojedynczych zmiennych. Odpowiednik `list()` po prawej stronie wyrażenia, czyli

`array()`, umożliwia tworzenie tablic z pojedynczych wartości. Przypisanie tablicy zwróconej przez funkcję `array()` do zmiennych w `list()` umożliwia zmianę kolejności tych wartości. Działa to również w przypadku więcej niż dwóch wartości:

```
list($yesterday,$today,$tomorrow) = array($today,$tomorrow,$yesterday);
```

Metoda ta nie jest szybsza niż zastosowanie zmiennych tymczasowych, dlatego należy stosować ją dla zachowania czytelności kodu, a nie zwiększenia jego szybkości.

Zobacz również

Dokumentacja funkcji `list()` jest dostępna pod adresem *http://www.php.net/list*, a funkcji `array()` — pod adresem *http://www.php.net/array*.

5.4. Tworzenie dynamicznej nazwy zmiennej

Problem

Chcemy dynamicznie utworzyć nazwę zmiennej. Na przykład chcemy zastosować nazwy zmiennych, które odpowiadają nazwom pól pochodzących z zapytania do bazy danych.

Rozwiązanie

Zastosujemy składnię zmiennej zmiennych PHP: wstawimy znak `$` przed zmienną, której wartość wyznacza nazwę zmiennej, którą chcemy utworzyć:

```
$animal = 'tygrysy';
$tygrysy = 103;
print $$animal;
103
```

Analiza

Wykonanie powyższego przykładu spowoduje wyświetlenie wartości `103`. Ponieważ `$animal = 'tygrysy'`, `$$animal` ma wartość `$tygrysy`, a jej wartością jest z kolei 103.

Stosując nawiasy klamrowe, możemy utworzyć bardziej skomplikowane wyrażenia, które będą wskazywać nazwy zmiennych:

```
$stooges = array('Moe','Larry','Curly');
$stooge_moe = 'Moses Horwitz';
$stooge_larry = 'Louis Feinberg';
$stooge_curly = 'Jerome Horwitz';

foreach ($stooges as $s) {
  print "$s naprawdę nazywa się ${'stooge_'.strtolower($s)}.\n";
}
Moe naprawdę nazywa się Moses Horwitz.
Larry naprawdę nazywa się Louis Feinberg.
Curly naprawdę nazywa się Jerome Horwitz.
```

PHP oblicza wyrażenie pomiędzy nawiasami klamrowymi i stosuje je jako nazwę zmiennej. Wyrażenie może nawet zawierać wywołania funkcji, na przykład `strtolower()`.

Zmienne zmiennych są także użyteczne przy przechodzeniu kolejno przez zmienne o podobnych nazwach. Załóżmy, że wysyłamy zapytanie do tabeli bazy danych, w której znajdują się pola `title_1`, `title_2` itd. Jeżeli chcemy sprawdzić, czy tytuł odpowiada którejkolwiek z tych wartości, najprostszym rozwiązaniem jest przejście przez nie kolejno w następujący sposób:

```
for ($i = 1; $i <= $n; $i++) {
    $t = "title_$i";
    if ($title == $$t) { /* jeśli się równa */ }
}
```

Oczywiście, pierwszym rozwiązaniem przychodzącym na myśl byłoby przechowywanie tych wartości w tablicy, ale jeżeli utrzymujemy stary kod, który wykorzystuje ten właśnie sposób (i nie możemy go zmienić), zmienne zmiennych okazują się wówczas pomocne.

Zastosowanie nawiasów klamrowych jest również niezbędne podczas wyjaśniania niejasności dotyczących elementów tablicy. Zmienna zmiennych `$$donkeys[12]` może mieć dwa znaczenia. Pierwsze to „odczytaj to, co znajduje się w dwunastym elemencie tablicy `$donkeys` i użyj tego jako nazwy zmiennej". Napiszemy wówczas `${$donkeys[12]}`. Drugie znaczenie to „zastosuj jako nazwę tablicy to, co znajduje się w skalarze `$donkeys` i poszukaj dwunastego elementu tej tablicy". Napiszemy wówczas `${$donkeys}[12]`.

Zobacz również

Dokumentacja zmiennych zmiennych jest dostępna pod adresem *http://www.php.net/lan➥guage.variables.variable*.

5.5. Stosowanie zmiennych statycznych

Problem

Chcemy zachować wartość zmiennej lokalnej pomiędzy kolejnymi wywołaniami funkcji.

Rozwiązanie

Zadeklarujemy zmienną jako statyczną przy użyciu słowa kluczowego `static`:

```
function track_times_called( ) {
    static $i = 0;
    $i++;
    return $i;
}
```

Analiza

Zadeklarowanie zmiennej jako `static` spowoduje, że jej wartość zostanie zapamiętana przez funkcję. Jeżeli więc funkcja zostanie wywołana w późniejszym czasie, nadal będziemy mogli uzyskać dostęp do wartości tej zapisanej zmiennej. Funkcja `pc_check_the_count()` z przykładu 5.1 wykorzystuje zmienne statyczne do zliczania liczby piłek odbitych i przepuszczonych przez baseballistę grającego na pozycji pałkarza.

Przykład 5.1. Funkcja pc_check_the_count()

```
function pc_check_the_count($pitch) {
    static $strikes = 0;
    static $balls   = 0;

    switch ($pitch) {
    case 'foul':
        if (2 == $strikes) break; // nic się nie dzieje, jeśli pitcher
                                  ↳nie uderzył 2 razy
        // w przeciwnym wypadku przejdź do przypadku 'strike'
    case 'strike':
        $strikes++;
        break;
    case 'ball':
        $balls++;
        break;
    }

    if (3 == $strikes) {
        $strikes = $balls = 0;
        return 'wyeliminowany';
    }
    if (4 == $balls) {
        $strikes = $balls = 0;
        return 'biegnie';
    }
    return 'uderza';
}

$what_happened = check_the_count($pitch);
```

W funkcji `pc_check_the_count()` to, co zdarzy się z pałkarzem, zależy od wartości licznika uderzeń `$pitch` znajdującego się wewnątrz funkcji w instrukcji `switch`. Moglibyśmy zamiast tego zwracać liczby piłek odbitych i przepuszczonych, lecz wymagałoby to także sprawdzania w różnych miejscach kodu, czy zawodnik powinien zostać wyeliminowany, czy ma biec do bazy, czy też pozostać na miejscu.

Zmienne statyczne zachowują swoje wartości pomiędzy kolejnymi wywołaniami funkcji, jednak wartość ta jest zapamiętywana tylko na czas pojedynczego wywołania skryptu. Zmienna zadeklarowana jako `static` dostępna w czasie jednego wywołania nie utrzyma swojej wartości do chwili kolejnego wywołania tej samej strony.

Zobacz również

Dokumentacja zmiennych statycznych jest dostępna pod adresem *http://www.php.net/lan*
➥*guage.variables.scope*.

5.6. Współdzielenie zmiennych pomiędzy procesami

Problem

Chcemy poznać sposób współdzielenie informacji pomiędzy procesami, aby zapewnić szybki dostęp do współdzielenie danych.

Rozwiązanie

Zapiszemy dane w segmencie pamięci wspólnej i zagwarantujemy wyłączny dostęp do niego za pomocą semafora:

```
$semaphore_id = 100;
$segment_id   = 200;
// pobiera uchwyt do semafora związanego z pożądanym
// segmentem pamięci wspólnej
$sem = sem_get($semaphore_id,1,0600);
// zapewnia sobie wyłączny dostęp do semafora
sem_acquire($sem) or die("Nie mogę przejąć semafora ");
// pobiera uchwyt do pożądanego segmentu pamięci wspólnej
$shm = shm_attach($segment_id,16384,0600);
// pozyskuje wartość z segmentu pamięci wspólnej
$population = shm_get_var($shm,'population');
// operuje na wartości
$population += ($births + $immigrants - $deaths - $emigrants);
// z powrotem zapisuje wartość w segmencie pamięci wspólnej
shm_put_var($shm,'population',$population);
// zwalnia uchwyt do segmentu pamięci wspólnej
shm_detach($shm);
// zwalnia semafor, by inne procesy mogły go pozyskiwać
sem_release($sem);
```

Analiza

Segment pamięci wspólnej jest częścią pamięci RAM w komputerze, do której dostęp mogą mieć różne procesy (na przykład większa liczba procesów serwera WWW, który obsługuje zapytania). Semafor sam zapewnia, że inne procesy nie zachodzą na siebie w momencie, gdy jeden z nich ma dostęp do segmentu pamięci wspólnej. Przed wykorzystaniem przez proces segmentu pamięci musi on przejąć sterowanie semaforem. Jeśli proces został już wykonany, semafor jest zwalniany po to, by umożliwić wykorzystanie go przez inny proces.

Aby przejąć sterowanie semaforem, należy zastosować funkcję `sem_get()`, która znajduje jego identyfikator. Pierwszym argumentem tej funkcji jest liczba całkowita będąca kluczem semafora. Kluczem może być jakakolwiek liczba całkowita pod warunkiem, że wszystkie programy, które będą miały dostęp do tego konkretnego semafora, stosują ten sam klucz. Jeżeli semafor z wyszególnionym kluczem jeszcze nie istnieje, zostaje on utworzony, ustawiana jest maksymalna liczba procesów, które mogą mieć dostęp do semafora, wskazywana przez drugi argument funkcji `sem_get()` (w tym przypadku jest to `1`),

a następnie ustawiane są uprawnienia semaforów wskazywane przez trzeci argument funkcji (`0600`). Uprawnienia te działają tak jak uprawnienia plików, tak więc `0600` oznacza, że użytkownik, który utworzył semafor, może go odczytywać oraz zapisywać do niego. Mówiąc o „użytkowniku", mamy na myśli nie tylko proces, który utworzył semafor, ale również każdy inny proces z tym samym identyfikatorem użytkownika. Uprawnienia o wartości `0600` powinny być wystarczające dla większości zastosowań, w których procesy serwera WWW działają jako procesy tego samego użytkownika.

Funkcja `sem_get()` zwraca identyfikator, który wskazuje na odpowiedni semafor systemu. Identyfikator ten należy zastosować do uzyskania kontroli nad semaforem za pomocą funkcji `sem_acquire()`. Funkcja ta czeka aż do momentu, w którym semafor będzie mógł zostać przejęty (na przykład wtedy, gdy inne procesy zwolnią semafor), po czym zwróci wartość `true`. W przypadku błędu zwrócona zostanie wartość `false`. Błędy mogą wynikać z nieprawidłowego określenia uprawnień, lub też mogą powstać, gdy ilość pamięci potrzebna do utworzenia semafora jest zbyt mała. Gdy tylko semafor zostanie przejęty, można odczytać zawartość segmentu pamięci wspólnej.

Najpierw za pomocą funkcji `shm_attach()` należy ustawić łącze do konkretnego segmentu pamięci wspólnej. Tak jak w przypadku funkcji `sem_get()`, pierwszym argumentem `shm_attach()` jest również klucz w postaci liczby całkowitej. Tym razem jednak identyfikuje on pożądany segment, a nie semafor. Jeżeli segment o wskazanym kluczu nie istnieje, zostanie on utworzony przy zastosowaniu pozostałych argumentów. Drugim argumentem (`16384`) jest rozmiar segmentu w bajtach, natomiast ostatni argument (`0600`) określa uprawnienia dostępu do segmentu. Wywołanie `shm_attach(200,16384,0600)` spowoduje utworzenie segmentu pamięci wspólnej o pojemności 16 kilobajtów, z którego odczyt i do którego zapis będzie mógł wykonywać tylko ten użytkownik, który go utworzył. Funkcja zwraca identyfikator potrzebny do odczytu i zapisu do segmentu pamięci wspólnej.

Po połączeniu z segmentem można z niego pobierać zmienne za pomocą funkcji `shm_get_var($shm, 'population')`. Funkcja ta przeszukuje segment pamięci wspólnej identyfikowany przez `$shm` i pobiera wartość zmiennej noszącej nazwę `population`. W pamięci wspólnej można przechowywać zmienne dowolnego typu. Gdy tylko zmienna zostanie pobrana, można na niej wykonywać operacje tak jak na innych zmiennych. Wywołanie funkcji `shm_put_var($shm,'population',$population)` spowoduje umieszczenie wartości zmiennej `$population` z powrotem w segmencie pamięci wspólnej jako zmiennej o nazwie `population`.

Zakończyliśmy już pracę z pamięcią wspólną. Należy się teraz z nią rozłączyć za pomocą funkcji `shm_detach()` i zwolnić semafor za pomocą funkcji `sem_release()`, aby mogły z niego korzystać inne procesy.

Główną zaletą pamięci wspólnej jest to, że działa ona bardzo szybko. Ponieważ jednak znajduje się ona w pamięci RAM, nie może pomieścić zbyt dużo danych, a poza tym dane te zostaną one utracone, jeżeli komputer zostanie zrestartowany (chyba że podjęte zostaną specjalne czynności zmierzające do zapisania informacji z pamięci wspólnej na dysk przed wyłączeniem komputera oraz ich ponownego załadowania do pamięci podczas uruchamiania). Poza tym pamięć wspólna nie jest dostępna w systemie Windows.

Zobacz również

Receptura 8.27 zawiera program, który korzysta z pamięci wspólnej; dokumentacja pamięci wspólnej oraz funkcji semafora jest dostępna pod adresem *http://www.php.net/sem*.

5.7. Enkapsulacja złożonych typów danych do postaci łańcucha znaków

Problem

Chcemy utworzyć łańcuch znaków reprezentujący tablicę lub obiekt w celu przechowywania go w pliku lub bazie danych. Łańcuch ten powinien umożliwiać łatwe przywrócenie tablicy lub obiektu w pierwotnej postaci.

Rozwiązanie

Zastosujemy funkcję `serialize()`, aby zakodować zmienne i ich wartości do postaci tekstowej:

```
$pantry = array('cukier' => '2 kg.','masło' => '3 kostki');
$fp = fopen('/tmp/spizarnia','w') or die ("Nie można otworzyć spiżarni");
fputs($fp,serialize($pantry));
fclose($fp);
```

Aby odtworzyć zmienne, zastosujemy funkcję `unserialize()`:

```
$new_pantry = unserialize(join('',file('/tmp/spizarnia')));
```

Analiza

Powstały po serializacji łańcuch znaków, który zostanie odtworzony jako zmienna `$pantry`, ma następującą postać:

```
a:2:{s:6:"cukier";s:5:"2 kg.";s:5:"masło";s:8:"3 kostki";}
```

Łańcuch ten przechowuje wystarczającą ilość informacji, aby przywrócić wszystkie wartości znajdujące się w tablicy, jednak sama nazwa zmiennej nie jest przechowywana w reprezentacji powstałej w wyniku serializacji.

Podczas przesyłania serializowanych danych w ramach adresu URL z jednej strony WWW na inną należy wywołać względem tych danych funkcję `urlencode()`, aby upewnić się, że znajdujące się w nich metaznaki URL zostaną właściwie obsłużone:

```
$shopping_cart = array('Bagietka z makiem' => 2,
                       'Zwykła bagietka' => 1,
                       'Lody' => 4);
print '<a href="next.php?cart='.urlencode(serialize($shopping_cart)).
'">Dalej</a>';
```

Ustawienia konfiguracyjne `magic_quotes_gpc` oraz `magic_quotes_runtime` mają wpływ na dane przekazywane do funkcji `unserialize()`. Jeżeli `magic_quotes_gpc` ma wartość `on`, dane przekazywane w adresach URL, zmienne `POST` lub *cookies* muszą przed deserializacją zostać poddane działaniu funkcji `stripslashes()`:

```
$new_cart = unserialize(stripslashes($cart)); // jeśli magic_quotes_gpc
                                                 ma wartość on
$new_cart = unserialize($cart);               // jeśli magic_quotes_gpc
                                                 ma wartość off
```

Jeżeli ustawienie `magic_quotes_runtime` ma wartość `on`, serializowane dane przechowywane w pliku muszą być podczas zapisu poddane działaniu funkcji `addslashes()`, a podczas odczytu — funkcji `stripslashes()`:

```
$fp = fopen('/tmp/cart,'w');
fputs($fp,addslashes(serialize($a)));
fclose($fp);

// jeśli magic_quotes_runtime ma wartość on
$new_cart = unserialize(stripslashes(join('',file('/tmp/cart'))));
// jeśli magic_quotes_runtime ma wartość off
$new_cart = unserialize(join('',file('/tmp/cart')));
```

Jeżeli `magic_quotes_runtime()` ma wartość `on`, wówczas również serializowane dane odczytywane z bazy danych muszą zostać poddane działaniu funkcji `stripslashes()`:

```
mysql_query(
    "INSERT INTO cart (id,data) VALUES (1,'".addslashes(serialize($cart))."')");

$r = mysql_query('SELECT data FROM cart WHERE id = 1');
$ob = mysql_fetch_object($r);
// jeśli magic_quotes_runtime ma wartość on
$new_cart = unserialize(stripslashes($ob->data));
// jeśli magic_quotes_runtime ma wartość off
$new_cart = unserialize($ob->data);
```

Serializowane dane przesyłane do bazy danych zawsze muszą zostać poddane działaniu funkcji `addslashes()` (lub innej odpowiadającej jej funkcji właściwej dla bazy danych), aby zapewnić, że zostaną one prawidłowo zapisane.

Zobacz również

Receptura 10.7 zawiera informacje dotyczące obsługi danych przesyłanych do bazy danych.

5.8. Wyświetlanie zawartości zmiennej w postaci łańcuchów znaków

Problem

Chcemy kontrolować wartości przechowywane w zmiennej. Może ona być skomplikowaną tablicą zagnieżdżoną lub obiektem, więc nie można tak po prostu jej wyświetlić ani przechodzić kolejno przez jej elementy.

Rozwiązanie

Zastosujemy funkcję `print_r()` lub `var_dump()`:

```
$array = array("name" => "Franek", 12, array(3, 4));

print_r($array);
Array
(
    [name] => Franek
    [0] => 12
    [1] => Array
        (
            [0] => 3
            [1] => 4
        )

)
var_dump($array);
array(3) {
  ["name"]=>
  string(6) "Franek"
  [0]=>
  int(12)
  [1]=>
  array(2) {
    [0]=>
    int(3)
    [1]=>
    int(4)
  }
}
```

Analiza

Dane wyświetlane przez funkcję `print_r()` są bardziej zwięzłe i łatwiejsze do odczytania. Natomiast dane wyświetlane przez funkcję `var_dump()` zawierają typ danych oraz długość każdej zmiennej.

Funkcje te przechodzą przez kolejne zmienne w sposób rekurencyjny, dlatego jeżeli w zmiennej znajduje się odwołanie wskazujące na nią samą, może to doprowadzić do powstania pętli nieskończonej. Na szczęście obie funkcje zawsze wstrzymują się przed wyświetlaniem bez końca informacji o zmiennych. Jeżeli funkcja `print_r()` napotka zmienną odczytaną już wcześniej, wyświetli łańcuch `*RECURSION*`, zamiast ponownie wyświetlać informacje o zmiennej i kontynuować przechodzenie kolejno przez resztę informacji, które zostały jeszcze do wyświetlenia. Jeżeli funkcja `var_dump()` napotka tę samą zmienną więcej niż trzy razy, wygeneruje błąd krytyczny i zakończy działanie skryptu. Rozważmy tablice `$user_1` i `$user_2`, które odwołują się nawzajem do siebie przez elementy `friend`:

```
$user_1 = array('name' => 'Maks Białystok',
                'username' => 'maks');

$user_2 = array('name' => 'Leon Blum',
                'username' => 'leon');
```

```
// Maks i Leon są przyjaciółmi
$user_2['friend'] = &$user_1;
$user_1['friend'] = &$user_2;

// Maks i Leon mają pracę
$user_1['job'] = 'Kanciarz';
$user_2['job'] = 'Księgowy';
```

Wywołanie funkcji `print_r($user_2)` spowoduje wyświetlenie następujących danych:

```
Array
(
    [name] => Leon Blum
    [username] => leon
    [friend] => Array
        (
            [name] => Maks Białystok
            [username] => maks
            [friend] => Array
                (
                    [name] => Leon Blum
                    [username] => leon
                    [friend] => Array
 *RECURSION*
                    [job] => Księgowy
                )

            [job] => Kanciarz
        )

    [job] => Księgowy
)
```

Jeżeli funkcja `print_r()` po raz drugi napotka odwołanie do `$user_1`, wówczas wyświetli łańcuch znaków `*RECURSION*` i nie zejdzie już dalej do tablicy. Następnie funkcja kontynuuje swoje działanie, wyświetlając pozostałe elementy tablic `$user_1` i `$user_2`.

W razie napotkania rekurencji funkcja `var_dump()` zachowuje się w inny sposób:

```
array(4) {
  ["name"]=>
  string(9) "Leon Blum"
  ["username"]=>
  string(4) "leon"
  ["friend"]=>
  &array(4) {
    ["name"]=>
    string(14) "Maks Białystok"
    ["username"]=>
    string(4) "maks"
    ["friend"]=>
    &array(4) {
      ["name"]=>
      string(9) "Leon Blum"
      ["username"]=>
      string(4) "leon"
      ["friend"]=>
      &array(4) {
        ["name"]=>
        string(14) "Maks Białystok"
        ["username"]=>
```

```
            string(4) "maks"
            ["friend"]=>
            &array(4) {
              ["name"]=>
              string(9) "Leon Blum"
              ["username"]=>
              string(4) "leon"
              ["friend"]=>
              &array(4) {
                ["name"]=>
                string(14) "Maks Białystok"
                ["username"]=>
                string(4) "maks"
                ["friend"]=>
                &array(4) {
                  ["name"]=>
                  string(9) "Leon Blum"
                  ["username"]=>
                  string(4) "leon"
                  ["friend"]=>
                  &array(4) {
<br />
<b>Fatal error</b>:  Nesting level too deep - recursive dependency?
↳in <b>var-dump.php</b> on line <b>15</b><br />
```

Zatrzymanie rekurencji nastąpi dopiero wówczas, gdy funkcja `var_dump()` napotka odwołanie do `$user_1` po raz czwarty. Gdy to nastąpi, funkcja wygeneruje błąd krytyczny i żadne inne wartości zmiennych nie zostaną wyświetlone ani nie zostanie wykonana żadna inna część skryptu.

Pomimo tego, że funkcje `print_r()` i `var_dump()` wyświetlają wyniki swego działania, a nie zwracają ich jako wartości zwracanej, dane te i tak można przechwycić, unikając ich drukowania dzięki zastosowaniu buforowania danych wyjściowych:

```
ob_start();
var_dump($user);
$dump = ob_get_contents();
ob_end_clean();
```

Wynik działania funkcji `var_dump($user)` zostanie umieszczony w zmiennej `$dump`.

Zobacz również

Receptura 8.12 opisuje buforowanie danych wyjściowych; receptura 10.8 przedstawia sposób obsługi błędów w module *PEAR DB* polegający na zastosowaniu buforowania danych wyjściowych funkcji `print_r()` w celu zachowania komunikatów błędów; dokumentacja funkcji `print_r()` jest dostępna pod adresem *http://www.php.net/print-r*, a dokumentacja funkcji `var_dump()` — pod adresem *http://www.php.net/var-dump*.