

Tomasz "Zyx" Jędrzejewski

Zarządzanie ścieżkami dostępu w skryptach PHP

Wersja 1.0 (29.03.2008)



Szczegółowe informacje o licencji znajdują się pod artykułem.

www.zyxist.com

Zarządzanie ścieżkami dostępu w skryptach PHP

Duże skrypty PHP składają się z dziesiątek lub nawet setek plików ułożonych w równie dużej liczbie katalogów. Aby zapanować nad tak dużą strukturą, potrzebujemy przemyślanego planu, dzięki któremu nie pogubimy się w tym wszystkim. W tym artykule jednak nie będę skupiał się na zasadach organizowania skryptu w pliki oraz rozmieszczania ich w folderach. Zamierzam omówić równie istotne zagadnienie – jak nauczyć skrypt PHP bezproblemowo odnajdywać potrzebne nam aktualnie pliki? Wielu początkujących ma z tym problem; sam także pamiętam swoje zmagania z dawnych lat – dojście do optymalnego i elastycznego rozwiązania zajęło mi trochę czasu.

Katalog roboczy

Aby odnaleźć jakiś plik na twardym dysku, musimy znać do niego ścieżkę dostępu pokazującą, jak z pewnego wyróżnionego punktu trafić do niego poprzez kolejne katalogi. We współczesnych systemach operacyjnych oraz interpreterach skryptów (także PHP) istnieją przynajmniej dwa takie punkty. Jednym z nich jest korzeń systemu plików. W klonach Uniksa, mamy słynny slash, od którego rozpoczynają się wszystkie ścieżki, natomiast w Windowsach - literę odpowiedniego dysku. Drugi punkt to katalog roboczy, zależny już od konkretnego programu, który analizujemy. Według założeń, program powinien wykonywać swoją pracę właśnie w nim. Nie zgłębiając się w szczegóły, dla skryptów PHP obsługujących strony WWW, najczęściej będzie to ten sam folder, w którym uruchomiony skrypt się znajduje. Oto prosty przykład: nasza strona siedzi sobie w **/home/mojserwis/www/**. Mamy tam plik **index.php** oraz katalog **foo**, gdzie znajduje się jeszcze jeden plik: **bar.php**. Uruchamiamy *localhost/index.php* oraz *localhost/foo/bar.php*. Jakie będą katalogi robocze ustawione dla każdego ze skryptów? Łatwo sprawdzić, że dla pierwszego z nich katalogiem głównym będzie **/home/mojserwis/www/**, zaś dla drugiego - **/home/mojserwis/www/foo/**.

Z oczywistych względów nasz skrypt nie może polegać na odnajdywaniu plików względem korzenia systemu plików, gdyż w ten sposób uzależnimy się od systemu operacyjnego. Co więcej, chociaż znamy układ katalogów na naszym dysku, nie możemy mieć pewności, w jaki sposób zorganizuje go sobie administrator właściwego serwera, który będzie obsługiwać naszą stronę. Jednak także katalog roboczy sprawia dużo kłopotów. Pokażę to na przykładzie. Utwórz sobie następującą strukturę katalogową gdzieś na swoim dysku twardym:

1. **data/**
2. **data/config.php**
3. **libs/library.php**
4. **www/script1.php**
5. **www/subdir/**
6. **www/subdir/script2.php**

Do pliku **config.php** wpisz cokolwiek, jego treść jest dla nas zupełnie nieistotna. **library.php** niech zawiera następujący kod:

```
<?php
    if(!file_exists('../data/config.php'))
    {
        echo 'Plik konfiguracyjny nie istnieje!';
    }
    else
    {
        echo 'Plik konfiguracyjny istnieje.';
    }
?>
```

Teraz w obu plikach **script1.php** oraz **script2.php** musimy dołączyć naszą „bibliotekę”:

```
<?php // script1.php
```

```
require('../libs/library.php');  
?>
```

```
<?php // script2.php  
require('../../libs/library.php');  
?>
```

Oczywiście pragniemy, aby nasza biblioteka zachowywała się tak samo bez względu na to, który z głównych plików ją dołączy: powinna w każdym przypadku wykryć obecność pliku konfiguracyjnego, wypisując stosowny komunikat. Jednak po uruchomieniu okazuje się, że tak nie jest. O ile **script1.php** wyświetla poprawnie „Plik konfiguracyjny istnieje.”, to **script2.php** atakuje nas komunikatem o braku takowego!

Popatrzmy, co dzieje się za kulisami. W PHP każda ścieżka zaczynająca się od **./** lub **../** oznacza, że chcemy poszukiwać pliku względem aktualnego katalogu roboczego. Zatem żeby **script1.php** odnalazł **library.php**, musimy najpierw wyskoczyć katalog wyżej (**../**), później zejść do **libs** i dopiero wtedy dostaniemy nasz plik. Ze **script2.php** musimy już wyskoczyć dwa katalogi wyżej. Aby z **libs.php** trafić do konfiguracji, musimy wyjść katalog wyżej i wejść do **data**. Jednak kończąc nasze rozumowanie w tym miejscu, omijamy kluczową informację: operacje **require()** oraz **include()** *nie zmieniają katalogu roboczego!* Wszystko zawsze odbywa się względem katalogu, w którym znajduje się uruchomiony skrypt główny. W przypadku **script1.php** przykład działa, ponieważ stąd również możemy dotrzeć do konfiguracji, skacząc raz do góry i wchodząc do **data**, a tak się składa, że dokładnie identyczna sekwencja kroków doprowadzi nas tam z biblioteki. Skrypt drugi nie ma tyle szczęścia. Skacząc do góry tylko raz, w rzeczywistości wylądujemy dopiero w **www/**, a tam żadnego katalogu **data** nie ma.

Postępując dalej w ten sposób nawet gdy pamiętamy o niezmiennianiu się katalogu roboczego, możemy wciąż zagonić się w ślepy zaułek. Typowy skrypt to znacznie więcej, niż trzy katalogi na krzyż z paroma plikami. Jeśli dalej będziemy próbowali pisać każdą ścieżkę na sztywno w kodzie, natknijemy się na następujące problemy:

1. Nasze biblioteki i moduły będą mogły pracować tylko na jednym ustawieniu katalogu roboczego. Jeżeli będzie chciał z nich skorzystać skrypt leżący np. w jakimś podkatalogu, nie będzie to możliwe.
2. Tworząc kod, musimy pamiętać o naszych zawirowaniach z katalogiem roboczym, a o tym można czasem z rozpędu zapomnieć i wstawić ścieżkę względem położenia biblioteki.
3. Praktycznie pozbawiamy się możliwości przeorganizowania struktury katalogowej. Próbuując zmienić nazwę jednego katalogu czy jego położenie, musimy wpiery odnaleźć w całym kodzie wszystkie korzystające z niego ścieżki i ręcznie je pozmienić.

Tak czy inaczej, jasne jest, że tak się skryptów nie pisze.

Zasady wstępne

Jeżeli coś ma działać dobrze, a nie udawać, że działa, musi mieć ręce i nogi, czyli jakieś ściśle oraz przemyślane zasady, według których pracuje. Również i my, aby zapanować nad chaosem ścieżek, spróbujemy sobie takie wynaleźć. Aby wysiłek miał sens, musimy na wstępie założyć, że opracowane rozwiązanie będzie przestrzegane w całym skrypcie.

Zastanówmy się najpierw, co jest główną wadą katalogu roboczego. Wszystkie nasze kłopoty wynikały z tego, że zmieniał się on w zależności od tego, gdzie leżał wywoływany skrypt. Gdy był to jakiś podkatalog, całość zaczynała się sypać, ponieważ żadna ścieżka nie uwzględniała dodatkowego skoku w górę, który powinna wykonywać, aby trafić tam, gdzie chcemy. Idealnym rozwiązaniem byłoby wyróżnienie jakiegoś katalogu i podawanie ścieżek względem niego – w sumie identycznie robią systemy plików na dysku twardym. Najlepiej do wyróżnienia nadaje się główny katalog projektu, w którym trzymamy wszystkie jego pliki. Jest to najwyższej położona lokalizacja i zawsze można z niej trafić wszędzie, schodząc wyłącznie w dół, bez wykonywania skoków do katalogu nadrzędnego. W naszym przytoczonym wyżej przykładzie wyróżniamy więc folder, w którym trzymamy **www**, **libs** oraz **data**. Prawidłową ścieżką jest więc **libs/library.php** bądź **data/config.php**, zaś **../libs/library.php** już nie, ponieważ pokazuje ona drogę z jednego z podkatalogów, a nie głównego.

Drugim ważnym krokiem będzie wyeliminowanie ścieżek z kodu modułów, bibliotek itd. Zakładamy, że te elementy skryptu nie muszą na starcie wiedzieć, gdzie znajdują się potrzebne im dane. Przecież te informacje mogą otrzymać z innego źródła już po uruchomieniu, zaś rola programisty ograniczy się tylko do wskazania, który rodzaj zasobów powinien je interesować. Następnym takim posunięciem jest konieczność napisania jakiegoś zarządcy, który tuż po uruchomieniu wygeneruje wszystkie ścieżki, bazując na aktualnych ustawieniach katalogu roboczego, a później umieści je w łatwo dostępnym miejscu. Wbrew pozorom, mechanizm ten jest wyjątkowo prosty. Wystarczy wykorzystać do jego stworzenia główny plik startowy skryptu (zwany często **bootstrap.php**, **common.php** lub podobnie), który w pierwszym kroku utworzy zbiór stałych z potrzebnymi ścieżkami, następnie załaduje niezbędne biblioteki, zainicjuje obiekty oraz zwróci sterowanie modułowi. Ponieważ plik ten odpalany jest zwykle przez inną część skryptu, a nie przez przeglądarkę, możemy wprowadzić ostatnie korekty i ostatecznie podsumować cały patent:

1. Plik PHP uruchamiany z przeglądarki musi jedynie znać swoje położenie względem głównego katalogu oraz ścieżkę do pliku startowego. Musi jakoś udostępnić te dane i uruchomić plik startowy.
2. Plik startowy zna położenie wszystkich elementów względem katalogu głównego. Na podstawie informacji o katalogu roboczym generuje wszystkie ścieżki, udostępnia je skryptowi oraz inicjuje działanie wszystkich obiektów.
3. Moduł, który może znajdować się zarówno w uruchomionym z przeglądarki pliku, jak też i w innym miejscu (rozwiązanie spotykane we frameworkach), nie wie nic o żadnych ścieżkach. Gdy potrzebuje odczytać jakiś plik, odwołuje się do odpowiednich stałych pokazujących, gdzie leży interesujący go zasób.

Zwróćmy uwagę na zalety tego rozwiązania:

1. Zmieniając organizację katalogów, musimy jedynie wyedytować definicje stałych w pliku startowym oraz przenieść pliki. Skrypt automatycznie dostosuje się już do nowych ustawień.
2. Moduły nie muszą martwić się o ścieżki. Mają bowiem gwarancję, że korzystając z odpowiednich stałych, plik startowy zadbał już o skierowanie ich we właściwe miejsce z *uwzględnieniem aktualnego katalogu roboczego*.

Przejdźmy teraz do implementacji.

Implementacja

Na podstawie utworzonej wyżej struktury katalogowej, pokażę teraz, jak w praktyce zaimplementować nasze rozwiązanie. Zaczniemy od jakiegoś pliku uruchamianego z przeglądarki. W naszym przypadku będą to **www/script1.php** oraz **www/foo/script2.php**. Muszą one na wstępie znać swoje położenie względem katalogu głównego i ścieżkę do pliku startowego, który należy odpalić. Zatem ich kod będzie następujący:

```
<?php // script1.php
define('DIR_MAIN', '../');
require(DIR_MAIN.'bootstrap.php');

echo 'To działa: '.odczytajCosZDysku();
?>
```

```
<?php // script2.php
define('DIR_MAIN', '../../');
require(DIR_MAIN.'bootstrap.php');

echo 'To także działa: '.odczytajCosZDysku();
?>
```

Funkcję *odczytajCosZDysku()* napiszemy później – pokaże nam ona doświadczalnie, że wszystko prawidłowo funkcjonuje. Przejdźmy teraz do pliku startowego, który zlokalizowany jest bezpośrednio w głównym katalogu. Z założenia nie można uruchomić go z przeglądarki – my zadaliśmy o to, po prostu umieszczając go w miejscu, do którego z przeglądarki nie da się dostać. Na początku, opierając się na stałej *DIR_MAIN*,

plik ten zadeklaruje wszystkie stałe ze ścieżkami, a później załaduje biblioteki i zainicjuje je.

```
<?php // bootstrap.php
define('DIR_WWW', DIR_MAIN.'www/');
define('DIR_FOO', DIR_WWW.'foo/');
define('DIR_LIBS', DIR_MAIN.'libs/');
define('DIR_DATA', DIR_MAIN.'data/');

require(DIR_LIBS.'library.php');

// tu można coś zainicjować, ale my nie mamy chwilowo co.
?>
```

Na koniec, aby PHP nie oburzył się, przepiszmy na nowo plik **library.php**, tworząc tam używaną w skryptach funkcję:

```
<?php
function odczytajCosZDysku()
{
    return file_get_contents(DIR_DATA.'config.php');
} // end odczytajCosZDysku();
?>
```

Skrypt możemy już uruchomić. Zauważymy, że tym razem nie występują takie problemy, jak poprzednio. Zarówno **script1.php**, jak i leżący w podkatalogu **script2.php** mają prawidłowy dostęp do niezbędnych zasobów, nawet wykorzystując te same biblioteki. Osiągnęliśmy to właśnie dzięki usunięciu z nich ścieżek oraz stworzeniu prostego, acz skutecznego zarządcy.

Stałe nie są jedynym rozwiązaniem, jeśli chodzi o przechowywanie listy ścieżek. Szczególnie we frameworkach łatwo natknąć się na tzw. *include_path*. Jest to ciąg tekstowy zawierający wszystkie potrzebne skryptowi ścieżki. Kiedy odwołujemy się do jakiegoś pliku poprzez wybrane funkcje, PHP próbuje odszukać go w podanych katalogach, po prostu sprawdzając je po kolei. W tej implementacji nieco inaczej wyglądać będą pliki **bootstrap.php** oraz **library.php**.

```
<?php // bootstrap.php
set_include_path(
    DIR_MAIN.'www/'. PATH_SEPARATOR.
    DIR_MAIN.'www/foo/'. PATH_SEPARATOR.
    DIR_MAIN.'libs/'. PATH_SEPARATOR.
    DIR_MAIN.'data/'. PATH_SEPARATOR
);
require('library.php');
?>
```

```
<?php // library.php
function odczytajCosZDysku()
{
    return file_get_contents('config.php', true);
} // end odczytajCosZDysku();
?>
```

Funkcja *set_include_path()* pozwala ustawić nam własną listę ścieżek do przeszukiwania na miejsce domyślnej. Ponieważ separator ścieżek jest zależny od systemu operacyjnego, na którym uruchamiamy skrypt, posiłkujemy się stałą *PATH_SEPARATOR*. W tym rozwiązaniu przy próbie dostępu do dysku znaczącemu skróceniu ulegają ścieżki. Podajemy już tylko nazwę interesującego nas pliku, natomiast PHP przeszukuje po kolei wszystkie katalogi aż do jego znalezienia.

Każdy kij ma jednak dwa końce. Zaletą stosowania *include_path* jest większa automatyzacja. Możemy bez przeszkód przenieść sobie plik w inne miejsce i nie martwić się w ogóle poprawianiem

jakichkolwiek ścieżek w samym skrypcie. Dopóki nowa lokalizacja znajduje się w obrębie znanych ścieżek, PHP na pewno ją odnajdzie bez naszej ingerencji. Płacimy za to niestety wydajnością. Operacje dyskowe są z definicji dość powolne, chociaż tutaj dość dużo zależy od systemu operacyjnego, systemu plików itd. Tak czy siak każdy szanujący się programista powinien dążyć do zminimalizowania ich liczby. Tymczasem tutaj robimy coś zupełnie odwrotnego. Załóżmy, że mamy dziesięć różnych ścieżek i próbujemy znaleźć plik, który znajduje się w dziewiątej. Aby go odczytać, PHP musi odpytać aż dziewięć katalogów, podczas gdy w rozwiązaniu ze stałymi od razu idzie tam, gdzie trzeba. Benchmarki pokazują, że działanie *include_path* jest szczególnie powolne, gdy próbujemy odwoływać się do nieistniejących plików.

Drugi problem charakter techniczny. Działanie *include_path* ograniczone jest tylko do pewnej grupy funkcji oraz konstrukcji języka (np. **require()** oraz **include()** korzystają z tego od razu, jeżeli nie rozpoczniemy naszej ścieżki od kropki lub wyskoku do nadrzędnego katalogu), a w wielu innych przypadkach trzeba ręcznie odpowiednim parametrem włączać żądanie jego wykorzystania. Jeżeli nie będziemy o tym pamiętać, funkcje takie, jak **fopen()** czy **file_get_contents()** będą pozbawione sensownego mechanizmu lokalizowania plików w naszym skrypcie! Ograniczenie się wyłącznie do *include_path* spowoduje, że nie będziemy mogli określić, gdzie chcemy zapisać jakieś dane na dysku. Niestety, w tym wypadku PHP nie wybierze już za nas katalogu.

Osobiście preferuję formułę ze stałymi. Co prawda wymaga ona trochę więcej ręcznego pisania, ale mi osobiście odpowiada to, że patrząc w kod mimo wszystko wiem, gdzie odczytywany przezeń plik *może* się znajdować. Jeśli przed jego nazwą jest stała *DIR_DATA*, wiem, że muszę zajrzeć do katalogu, w którym są dane.

Zakończenie

Okazuje się, że nawet zwyczajne zarządzanie ścieżkami wymaga odpowiedniego podejścia. Bagatelizowanie tej sprawy zemści się na nas już w niedalekiej przyszłości, kiedy będziemy tracili coraz więcej czasu na błędy związane z gubieniem się w istniejących już ścieżkach oraz katalogach roboczych. Możemy tego prosto uniknąć, wprowadzając na samym początku pewne przemyślane reguły i ściśle się ich trzymając. Podany w artykule sposób nie jest jedynym na rozwiązanie wspomnianego problemu, ale z pewnością skutecznym. Stosuję go już od kilku lat i nie widzę żadnej potrzeby zastępowania go czym innym.

Tomasz "Zyx" Jędrzejewski

Aktualna wersja artykułu zawsze na **www.zyxist.com**

Licencja

Artykuł rozpowszechniany jest na licencji **Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów zależnych 2.5 Polska**.

Wolno:

- kopiować, rozpowszechniać, odtwarzać i wykonywać utwór

Na następujących warunkach:

1. *Uznanie autorstwa*. Utwór należy oznaczyć w sposób określony przez Twórcę lub Licencjodawcę*.
2. *Użycie niekomercyjne*. Nie wolno używać tego utworu do celów komercyjnych.
3. *Bez utworów zależnych*. Nie wolno zmieniać, przekształcać ani tworzyć nowych dzieł na podstawie tego utworu.

W celu ponownego użycia utworu lub rozpowszechniania utworu należy wyjaśnić innym warunki licencji, na której udostępnia się utwór.

Każdy z tych warunków może zostać uchylony, jeśli uzyska się zezwolenie właściciela praw autorskich.

Powyższe postanowienia w żaden sposób nie naruszają uprawnień wynikających z dozwolonego użytku ani żadnych innych praw.

Internetowa skrócona wersja licencji:

<http://creativecommons.org/licenses/by-nc-nd/2.5/pl/>

Pełen tekst licencji:

<http://creativecommons.org/licenses/by-nc-nd/2.5/pl/legalcode>

Podczas publikacji należy podać następujące informacje:

1. Link do internetowej skróconej wersji licencji.
2. Informację o wersji publikowanego tekstu.
3. Informacje o autorze w następujący sposób:

Tomasz "Zyx" Jędrzejewski

Aktualna wersja artykułu zawsze na **www.zyxist.com**