PHP jeszcze do niedawna nie miał wcale kontroli typów, teraz to się już trochę zmieniło . Ja jednak uważam, że dla języka skryptowego, który nie jest kompilowany są lepsze sposoby na "wymuszenie" typu niż podawanie go jawnie. Jak? Zapraszam do lektury.

Na początek

- ogólna wiedza o OO PHP5
- serwer www, np. Wamp

Problem z typami

PHP jak to język skryptowy nie posiada silnej typizacji. Do zmiennej można przypisać cokolwiek i nie powoduje to błędu. To, że nie powioduje to błędu składniowego to dobrze. Ale co z logiką. przykładowy kod:

{codecitation class='brush: php'}

class ExampleClass { public $value = 'moja wartość'; } function example($param) { echo $param->value; } example(new ExampleClass()); // to zadziala example("To nie jest obiekt"); // tu bedzie blad

{/codecitation}

W wyniku działania takiego skryptu pojawi się komunikat:

{codecitation}moja wartość
Notice: Trying to get property of non-object in **C:wampwwwlocalhosta.php** on line **8**{/codecitation}

niby nie błąd, ale przecież ten skrypt działa **nieprawidłowo**. Skoro oczekiwaliśmy jakiejś wartości w tym miejscu, to ona powinna wystąpić.

Łata goni łatę

PHP mimo, że nie posiada silnej typizacji, rozpoznaje typy zmiennych. Pozwala to sprawdzić, czy oby podane dane na pewno są prawidłowe (czyli takie, jakich oczekiwaliśmy). Oto zestaw funkcji, których można użyć, aby sprawdzić, czy dane są poprawne:

- bool is_callable (callback $name [, bool $syntax_only = false [, string &$callable_name]])
- bool is_array (mixed $var)
- bool is_numeric (mixed $var)
- bool is_int (mixed $var)
- bool is_object (mixed $var)
- bool is_a (object $object , string $class_name)
- bool is_string (mixed $var)
- bool is_float (mixed $var)
- bool is_null (mixed $var)

Każda z tych funkcji może być użyta, aby sprawdzić, czy podany parametr jest odpowiedniego typu. Gdybym chciał poprawić powyższy kod, mógłbym zrobić:

{codecitation class='brush: php'}

function example($param) { if (is_a($param, 'ExampleClass')) { echo $param->value; } else { echo 'Nie ma takiej wartości'; } } example(new ExampleClass()); // wyswietli: 'moja wartość' example("To nie jest obiekt"); // wyswietli: 'Nie ma takiej wartości'

{/codecitation}

Jednak taki if wewnątrz funkcji troche niepotrzebnie nam wydłuża kod. Jak wcześniej wspomniałem w najnowszych wersjach PHP mamy możliwość zrobienia tego ładniej.

No dobrze, pojawił nie pojawi sie nam komuniakat (który można zresztą wyłączyć). Ale nadal nie było tam wartości, jakiej oczekiwaliśmy. Programowanie wymaga konkretów i ścisłości. Jeśli nie ma jakichś danych, powinno to oznaczać błędne i nieporządane działanie. Działanie szkodliwe i potencjalnie niebezpieczne. A takie działanie powinno wywoływać błąd, o czym za chwilę.

Kontrola typów w PHP - mechanizm wbudowany

W PHP 5 pojawiła się kontrola typu obiektu, a w PHP5.1 pojawiła się możliwość wymuszenia, by przekazany parametr był tablicą. Prosty przykład:

{codecitation class='brush: php'}

function example(ExampleClass $param) { // dla tablicy: example(array $param) echo $param->value; } example(new ExampleClass); // to zadziala example("To nie jest obiekt"); // tu bedzie bla

{/codecitation}

Powyższy kod jest trochę czytelniejszy. Przypomina nawet troche kody języków C-podobnych. W wyniku uruchomienia dostajemy taki komunikat:

{codecitation}moja wartość
Catchable fatal error: Argument 1 passed to example() must be an instance of ExampleClass, string given, called in C:wampwwwlocalhostexample.php on line 13 and defined in **C:wampwwwlocalhostexample.php** on line **6**{/codecitation}

No i mamy to, czego chciałem. Błędna dane = błąd = koniec skryptu. Ale teraz powstaje problem, bo być może chcielibyśmy sobie jeszcze wysłać informacje o tym, co sie stało na maila. A może musimy jescze zamknąć połączenie z bazą danych, lub skasowac jakiś plik... Co prawda - błędy można przechwycić za pomocą funckji set_error_handler :

{codecitation class='brush: php'}mixed set_error_handler (callback $error_handler [, int $error_types = E_ALL | E_STRICT]){/codecitation}

Mi jednak na myśl o takich zabiegach nie zgadzają się drobne w kieszeni. Przecież mamy wyjątki. Dlaczego nie zamienić naszego podejścia do skryptu i zamiast wywoływać błędy - rzucać wyjątki. Które są czytelniejsze, pozwalają uzyskać wiele informacji w bardziej przyjazny sposób.

W przypadku języków kompilowanych takie błędy byłyby wykryte na etapie kompilacji - przed uruchomieniem. Niestety w PHP zostaną wykryte dopiero w trakcie działania. To na pewno jest dosyć uporczywe. Co więcej w ten sposób można wykrywać jedynie obiekty lub tablice. Nie ma możliwości, aby rozpoznać typy proste (ciąg znaków, liczbę całkowitą lub zmiennoprzecinkową).

Run-time type hinting

Czyli sprawdzanie typu w locie. Jest właściwy typ - super, działamy. Jest niewłaściwy - trudno, kończymy zabawę.

Na początek potrzebujemy klasy. Tak sobie myślę, że można uznać, że nie tylko klasy. Pakietu (jeśli nie wiesz jak tworzyć "pakiety" w PHP < 5.3.3 to zapraszam do lektury)! nazwijmy ten pakiet `Validation'. Czyli w folderze `Validation' tworzymy sobie plik klasy o nazwie `Type.php'. Oznacza to, że nasza klasa nazywać sie będzie `Validation_Type':

{codecitation class='brush: php'}

class Validation_Type { const MSG = 'Wymagany typ %s. Podano %s.'; static protected function message($exp, $passed) { return sprintf(self::MSG, $exp, $passed); } static public function isNull($data) { if (!is_null($data)) { throw new Validation_Exception_NullExpected(self::message('null', gettype($data))); } return true; } // pelny kod klasy umieszczam na SVN }

{/codecitation}

Pełny kod klasy na svn: http://php-validation.googlecode.com/svn/trunk/Type.php

Oraz klasy wyjątków, będące "podpakietem" - przydatne przy odpowiednim użyciu autoloadera. Przykładowa klasa wygląda tak:

{codecitation class='brush: php'}class Validation_Exception_NullExpected extends Validation_Exception_Type {}{/codecitation}

Kody pozostałych klasy dostępne na SVN: http://php-validation.googlecode.com/svn/trunk/Exception/

Wykorzystanie

{codecitation class='brush: php'}

// tu odpowiedni autoloader lub wiele plikow zalaczonych

function example($obj, $str, $n) { Validation_Type::is($obj, 'ExampleClass'); Validation_Type::isNotEmptyString($param); Validation_Type::isInteger($n); // tu wtedy cos wykonaj } example(new ExampleClass, 'To jest napis', 12); // to zadziala example(new

ExampleClass, 'To jest napis', 12.0); // to nie zadziala

{/codecitation}

Wynik działania:

{codecitation}**Fatal error**: Uncaught exception 'Validation_Exception_IntExpected' with message 'Wymagany typ int. Podano double.' in C:wampwwwlocalhosttypeValidationType.php:38 Stack trace: #0 C:wampwwwlocalhosttypeindex.php(20): Validation_Type::isInteger(12) #1 C:wampwwwlocalhosttypeindex.php(24): example(Object(ExampleClass), 'To jest napis', 12) #2 {main} thrown in **C:wampwwwlocalhosttypeValidationType.php** on line **38**
{/codecitation}

Dlaczego? otóż 12.0 to nie jest liczba całkowita.

Zalety?

Moim zdaniem takie rozwiązanie ma wiele zalet, np. to że możemy taki błąd obsłużyć w wielu miejscach. Choćby:

Elastyczna obsługa błędów

{codecitation class='brush: php'}

function example($obj, $str, $n) { Validation_Type::is($obj, 'ExampleClass'); Validation_Type::isNotEmptyString($param); try { Validation_Type::isInteger($n); } catch (Validation_Exception_IntExpected $e) { // poinformuj, ze cos bylo nie tak, ale dzialaj dalej $n = intval($n); // rzutujemy } // tu wtedy cos wykonaj } example(new ExampleClass, 'To jest napis', 12); // to zadziala example(new ExampleClass, 'To jest napis', 12.0); // to zadziala

{/codecitation}

I dzięki temu możemy już w funkcji przechwycić niektóre błędy. Tak jak na powyższym listingu. Niepoprawne dane zostały zwalidowane i doprowadzone do stanu, w ktorym nie szkodzą. Równie dobrze można zrobić tak:

{codecitation class='brush: php'}

function example($obj, $str, $n) { Validation_Type::is($obj, 'ExampleClass'); // 1 Validation_Type::isNotEmptyString($param); // 2 Validation_Type::isInteger($n); // 3 // tu wtedy cos wykonaj } example(new ExampleClass, 'To jest napis', 12); // to zadziala try { example(new ExampleClass, 'To jest napis', 12.0); // to zadziala } catch (Validation_Exception_IntExpected $e) { echo 'Nie udało się wykonac akcji. Błędne dane. Proszę uzupełnić formularz ponownie'; }

{/codecitation}

lub też przechwycić to jeszcze wyżej obejmując cały kod w jedne try catch i wyświetlac jedynie informacje o błędnym działaniu systemu, a do logga systemowego lub na maila wysłać sobie dokładną informację o tym, co było nie tak.

Wg mnie jest to rozsądne w sytuacji, kiedy takie błędy ujawniają się w trakcie działania (jak już mówiłem i pewnie wiesz, PHP nie kompiluje się).

Wymuszenie walidacji

Takie podejście wymusza także walidację. Jeśli skrypt nam sie wyłoży, gdy przekażemy nieprawidłowe dane, to będziemy bardzo dbali o to, aby dane te były prawidłowe. I - w co bardzo chcę wierzyć - taki kod nie pojawi się więcej:

{codecitation class='brush: php'}

example(new ExampleClass(), $_GET['napis'], $_SESSION['user_store']['data']['newDate']); // to zadziala example(new ExampleClass(), $_SESSION['user'], $_POST['tmp']); // to zadziala

{/codecitation}

Oczywiście to przykład bardzo niewłaściwie napisanego kodu. Choćby samo używanie zmienych superglobalnych jest ubogie*, a do tego dochodzi jeszcze używanie bardzo rozbudowanych struktur, jak choćby `$_SESSION['user_store']['data']['newDate']', bez sprawdzenia czy tamte dane istnieją. Polecam w tych sytuacjach konstrukcje językowe isset / empty
.

Trochę lepiej ten kod wyglądałby tak:

{codecitation class='brush: php'}

example(new ExampleClass(), strval($_GET['napis']), intval($_SESSION['user_store']['data']['newDate'])); // to zadziala example(new ExampleClass(), strval($_SESSION['user']), intval($_POST['tmp'])); // to zadziala

{/codecitation}

Funckje intval , strval , floatval itp. służą do jawnego wymuszenia rzutowania do danego typu. Więcej o nich możesz przeczytać w manualu.

Oczywiście lepiej byłoby w powyższym kodzie zwalidowac te dane poprawniej. Jednak już tylko taka walidacja (szczególnie zrzutowanie do intów) pozwala nam się zabezpieczyć przed kilkoma błędami. Więcej o walidacji wartości napiszę w przyszłości.

--
tak można ich używać, ale należy zawsze po przechwyceniu danych najpierw je zwalidować i działac już na poprawnych danych.