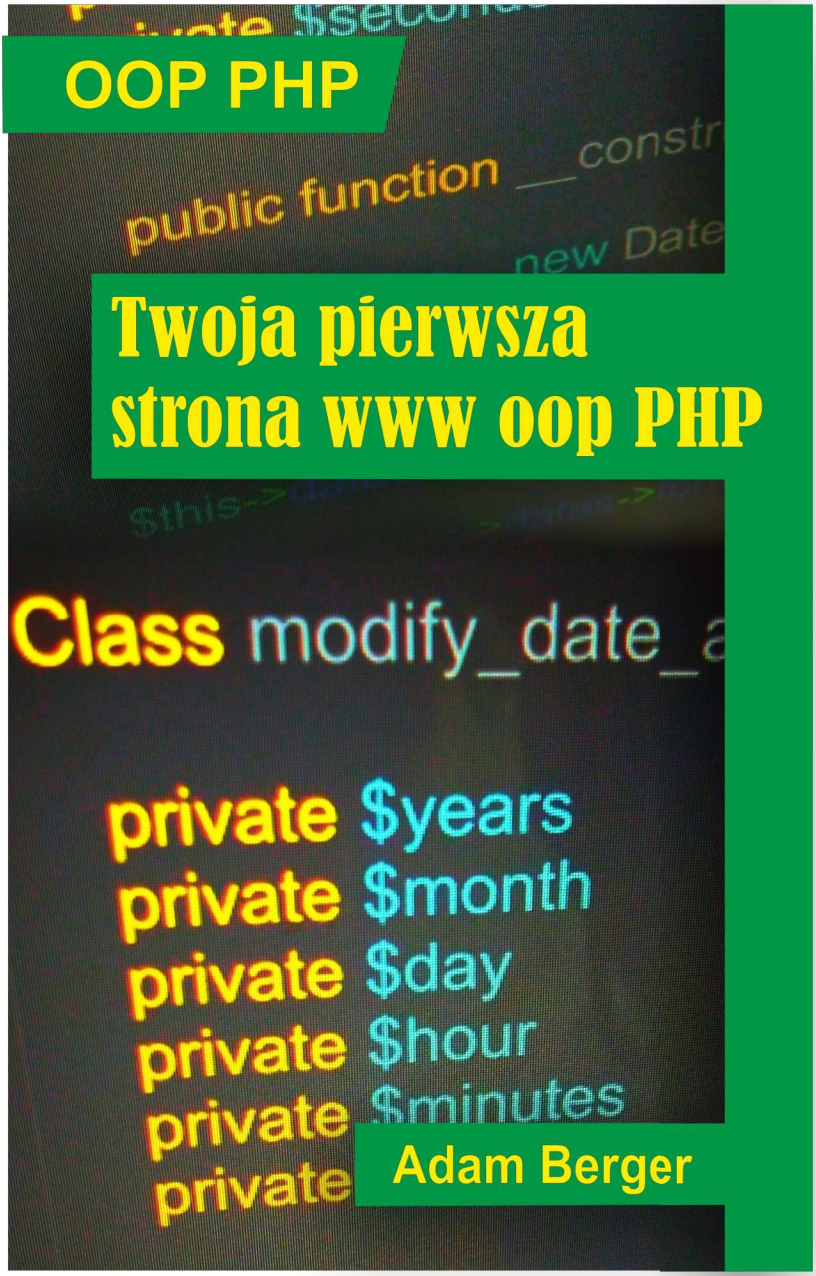

OOP PHP
public function __constr
new Date
Twoja pierwsza
strona www oop PHP
$this->
Class modify_date_a
private $years
private $month
private $day
private $hour
private $minutes
private
Adam Berger

Twoja pierwsza strona www OOP PHP

Skład: Adam Berger
Projekt Okładki: Adam Berger
Kontakt z autorem: adam.berger.developer@gmail.com

ISBN: 978-83-938738-2-1

Wstęp

Twoja pierwsza strona **www** postawiona w środowisku **OOP PHP** korzystająca z klas i metod, interfejsów klas abstrakcyjnych i obsługująca połączenie z bazą danych za pomocą PDO.

Pokażę wam jak w łatwy sposób można wykonać serwis internetowy, który zawiera rejestrację, logowanie, przypomnienie hasła, a także formularz kontaktowy z powiadomieniem na email. Przypiszemy też logi globalne i użytkownika z wysłaniem na e-mail lub nie w zależności, które dane będziemy chcieli uzyskać w zapisie do pliku.**log**. Wszystko będzie się opierało o najnowszy **html5**, **php 5**>, **jQuery**, javascript, css, pdo, php**, oop** i oczywiście klasach. Zademonstruję na przykładach użycie wzorców projektowych **prototype** i **special arguments**.

Skrypt, jaki wykonamy będzie mógł stanowić bazę pod wasze przyszłe projekty, który z czasem będziecie mogli rozbudować o rzeczy potrzebne na stronie.

Dlaczego właśnie w taki sposób chcę wam przekazać moją wiedzę, którą zdobyłem, a nie polecać wam uczenie się na samym początku jakiegoś Framework, bo wydaje mi się, że w ten sposób będzie wam łatwiej zrozumieć zagadnienia i same działa-

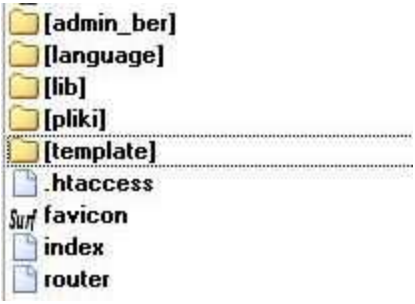
nie związane z **oop**, klasami i metodami niż właśnie używając samego **Frameworka**?

Żeby nie było nie porozumień w tym temacie, Frameworki są potrzebne i można się ich uczyć jak ktoś ma na to ochotę, lecz na początek proponuję zacząć od przeczytania mojej książki, która przedstawia tą kwestię jakby od środka zobaczysz same działanie klas, gdzie w **Frameworkach** raczej nie będziesz się zagłębiał jak np. działa walidacja formularza wystarczy, że wiesz, że dane z tablicy **$_POST** masz dać do metody x itd. A tu poznasz to od środka od samej budowy kodu po wywołanie samych klas.

Taki sposób nauki według mnie jest bardziej efektywny i jeżeli już wiesz, na czym to polega uczenie się **Frameworka** nie będzie ci sprawiało trudności, bo będziesz wiedział, co w metodzie x się dzieje podczas walidacji naszego przykładowego formularza.

Katalogi

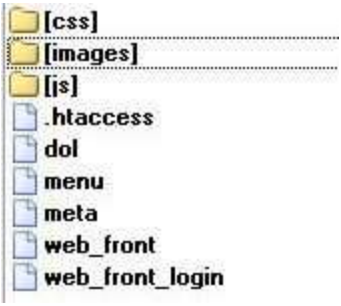
Zacznijmy na początek od rozplanowania samych plików, co gdzie ma być poumieszczane tzw. Drzewko katalogów



Rysunek 1.0 Drzewo katalogów

Pierwszy katalog będzie zawierał pliki **panelu administratora** możemy go nazwać tak jak chcemy, w drugim **language** umieścimy plik z tłumaczeniami dla naszej strony, będzie w nim wszystko, co będzie wymagało tłumaczenia, w naszym przypadku będzie to język polski i angielski, ale można rozbudować skrypt o dowolną liczbę języków. Następny katalog to **lib** w nim umieścimy przygotowane klasy do prawidłowego działania naszej strony WWW.

Katalog **pliki** będzie służył do przechowywania naszych plików użytkowych takich jak np. logowanie. **Template** jak sama nazwa wskazuje będą w nim umieszczone pliki i katalogi związane z wyglądem naszej witryny obrazek 1.1 przedstawia zawartość katalogu **template**.



Rysunek 1.1 katalog template

katalog **template /Css** posłuży do przechowywania pliku stylów (wyglądu) .css
images do zdjęć
js do plików .js
rozbiłem całą stronę na części żeby nam było łatwiej ingerować w kod strony, według mnie takie podejście do wyglądu jest łatwiejsze do ogarnięci w późniejszym etapie projektu.

Czyli tak **Web_front** to strona dla niezalogowanych użytkowników, a **Web_front_login** to wygląd dla zalogowanych użytkowników. **Dol** to stopka naszego szablonu, **menu** wiadomo i **meta**, dlatego że dość często wprowadza się tam zmiany, bo to jakiś slider się dokłada, albo trzeba zaktualizować **jQuery** itd.…

Co sądzicie o takim podziale, z pewnością na razie nic, bo nie wiecie, co zawierają owe pliki?

Po zobaczeniu zawartości przekonacie się o zasadności takiego rozwiązania.

Został nam jeszcze **index.php**, który będzie odpowiedzialny za całość, **router.php** w nim ustawimy automatyczne wczytywanie klas, z funkcji **PHP spl_autoload_register()** , w ten sposób będziemy mieli do nich dostęp wszędzie w całym projekcie.
Plik **.htaccess** pomoże nam zamienić linki na przyjazne wyszukiwarkom i zabezpieczyć dane.
I jeszcze ikonka dla przeglądarki **favicon.ico**.

Dobrym zwyczajem jest też opisywanie obiektów i zmiennych wykorzystywanych w naszym projekcie, żeby wiedzieć, które i jakie dane trzeba dodać zaimportować itd.
Jeden przykład zademonstruję, o co mi chodzi.

```
/**
* zbior()
* Sting  $string
* bolea $bolea
*/

$string=”asdasasdasdasdad”;
$bolea= true;

public function zbior ($string, $bolea){

}
```

Nie będę celowo w naszym przykładowym skrypcie robił opisów do poszczególnych klas i metod te zadanie pozostawiam wam. Będzie wam łatwiej zrozumieć jak działa ten mechanizm i nabierzecie pewnego nawyku opisywania klas i metod, co w późniejszych waszych projektach zaowocuje zrobioną porządną dokumentacją do waszego skryptu. Będę zamiast tego zamieszczał informację o działaniu pewnych mechanizmów, żeby było łatwiej zrozumieć.

1 Klasy

1.1 Rozmieszczenie Klas

Zastanawiałem się, od jakiej klasy zacząć i doszedłem do wniosku, że klasa sesji będzie najbardziej odpowiednia, ponieważ bez sesji nie będziemy mogli korzystać z witryny.

Rozdzielimy klasy na abstrakcyjne i normalne, dodamy także jakieś interfejsy. Zastanówmy się teraz, które klasy mają zostać abstrakcyjne i dla czego. Moje zdanie jest takie, że połączenie z bazą danych może zostać klasą abstrakcji, dlatego że nie musimy tworzyć dla tej klasy żadnych obiektów, możemy od-

woływać się bezpośrednio do potrzebnej metody i tak właśnie postąpiłem w klasie odpowiedzialnej za połączenie z bazą danych.

Następną klasą abstrakcyjną, jaką postanowiłem zrobić to klasa odpowiedzialna za wczytywanie odpowiednich klas dla projektu nazwałem ją **ClassAutoload:: .**

Kolejną klasą abstrakcyjną jest **errorLogiClass::.** Jak sami zauważyliście, zapisuję je w inny sposób niż standardowe klasy do tworzenia obiektów, w innym przypadku byłby zwrócony błąd jest to podyktowane właśnie tym, że są abstrakcyjne.

Podczas początkowego planowania zastanawiamy się jak całość ma wyglądać i jakie klasy będą nam potrzebne i które metody wykorzystamy np. w dwóch lub więcej klasach.

Pamiętajmy także o stawiania innych rozwiązań nad dziedziczeniem, ponieważ wiąże nam to ręce, co do dalszego postępowania z daną klasą, w szczególności teraz gdzie **PHP 5** daje większe możliwości w projektowaniu **OOP**.

1.2 Klasa abstrakt databaseClass połączenia z bazą danych & interface & metod static

Połączenie z bazą danych będzie odbywało się za pomocą mysql i pdo. Przydzieliłem nazwę **databaseClass** i plik tak samo **databaseClass.php.** Określimy zmienną na **prywatne, dlatego** że, nie będziemy ich nigdzie poza samym połączeniem potrzebować i używać.

Właśnie w tych kontenerach przypiszemy nasze dane do połączenia takie jak port, host, nazwa bazy danych, nazwa użytkownika, i hasło. Określimy także prefiks dla każdej tabeli w bazie danych. Co jest niejako zabezpieczeniem przed **hacker**.

```
interface interf_db{
  static public function polacz();
  static public function dbprefix();
}
abstract class databaseClass implements interf_db{

        static private $engine   = 'mysql';
        static private $host     = 'localhost'; // 127.0.0.1
        static private $port     = 3306;
        static private $database = 'ksiazka';
        static private $user     = 'root';
        static private $pass     = [redacted];
        static private $dns;
        static private $DbPrefix = 'prk_';
        static private $pdo;
```

Rysunek 1.2 połączenie baza danych

Same połączenie zaczynamy od bloku chronionego:

```
try{
```

a kończymy na:

```
catch(PDOException $e){
   echo 'Połączenie nie mogło zostać utworzo-
ne.<br />
```

```
            '.$e->getMessage().'<br />
            '.$e->getCode().'<br />
            '.$e->getFile().'<br />
            '.$e->getTrace().'<br />
            '.$e->getLine().'<br />
            '.$e->getPrevious();
    }
```

Dlaczego tyle tu metod błędów, ano, dlatego żeby nam było łatwiej zlokalizować ewentualny błąd połączenia albo inny. **Pdo** daje nam możliwość wywołania wszystkich na raz i jest to super sprawa. Poznajemy w nim wiadomość błędu, kod błędu, plik błędu, w której linii itd. Jest to wspaniały mechanizm pomocy programistom programujący właśnie w tej technice. Sami się przekonacie jak będziecie używać **PDO** chyba, że już używacie J.

Dziedziczymy w tej klasie po klasie **PDO**, więc możemy utworzyć metodę **public** i w niej samo połączenie. Przypiszemy dane do klasy **new PDO(nasze dane)** wcześniej przygotowane takie jak dns, użytkownik i hasło.

```
static private function polacz_tu(){
   try{
     if(!empty(self::$database)){
       self::$dns = self::$engine.':host='.self::$host.';port='.self::$port.';'
       . 'dbname='.self::$database.';';
       self::$pdo = new PDO(self::$dns, self::$user, self::$pass, array(
            PDO::ATTR_PERSISTENT => true,
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\'',
            PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true));  // bufor 1-50MB

        self::$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        return self::$pdo;
        }
     }catch(PDOException $e){
        $mesage1= "";
     echo   $mesage1 .= '<br>Połaczenie Pdo nie mogło zostać utworzone.<br />'
        .$e->getMessage().'<br />'
        .$e->getCode().'<br />'
        .$e->getFile().'<br />'
        //.$e->getTrace().'<br />'
        .$e->getLine().'<br />'
        .$e->getPrevious();
          }
          ## Pobieramy Super Globalna ##
        errorLogiClass::set_serwer($_SERVER);
          ## Zapis naszych logów ##
          $html=true;
          errorLogiClass::global_logi_email($mesage1, $html);
          errorLogiClass::global_logi_save($mesage1);
   }
```

Rysunek 1.3 połączenie New PDO