A tíz tuti tipp a PHP-val ismerkedők számára

Tíz olyan javaslatról lesz szó, amelyek segíthetnek elkerülni a leggyakoribb buktatókat PHP webalkalmazások készítése közben.

Aligha kétséges, hogy a *PHP* az egyik legkönnyebben használható programnyelv kezdők számára dinamikus web oldalak készítéséhez. A *PHP*, *Linux*, *Apache*, *MySQL* összeállítás olyannyira népszerű, hogy már betűszó vált belőle: *LAMP* (*Linux*, *Apache*, *MySQL* és *PHP*). Számos weboldal születik, anélkül, hogy a készítőjének bármit be kellene állítania, vagy programoznia ehhez. Egyszerűen csak keresnek egy előre elkészített kódrészletet valamelyik keresővel, beillesztik a *HTML* sablonba, feltöltenek mindent a webkiszolgálójukra, és készen vannak.
Illetve csak hisszük, hogy így van. Még az előzetes programozói tapasztalat sem sokat segít, mivel a gépre és a webre történő alkalmazáskészítés két külön világ. Ennek köszönhetően, amikor az emberek bemásolnak egy *PHP* kódrészletet, gyakran semmi sem történik (legalábbis semmi jó). Az oldalak lassan vagy rosszul töltődnek be, a programozó által kiszemelt kódrészlet pedig újabb biztonsági rést nyit.
Az itt leírt tippeket különösen azoknak ajánlom, akik tisztában vannak a programozás alapjaival, de még soha nem találkoztak a *PHP*-val.
A javaslatok nagyjából három kategóriába sorolhatók: hogyan fogjunk hozzá helyesen, hogyan ne okozzunk magunknak kárt, és végül hogyan tegyük a kódunkat hatékonyabbá. Egyrészt terjedelmi okok miatt, másrészt azért, mert számtalan kiváló on-line és papír alapú dokumentáció létezik, a legtöbb tipp azt mutatja meg, hogy mire figyeljünk, és miért.

1. Ellenőrizzük, hogy mindent helyesen telepítettünk és beállítottunk

A kezdeti megdöbbenés egyik gyakori forrása, hogy amikor a kezdő *PHP*-s feltölti az első weblapját valamelyik kiszolgálóra, a *PHP*/*HTML* forrást látja a böngészőjében a program kimeneti eredménye helyett. Ez azért van, mert a webkiszolgáló nem ismeri fel a *PHP* fájlunkat, így nem továbbítja azt a *PHP* értelmezőhöz. Ennek az oka, hogy a rendszergazda elfelejtette a *PHP* értelmezőhöz társítani a *PHP* fájlokat. A mulasztást az Apache beállítási fájlban, vagy a weboldal mellett elhelyezett *.htaccess* fájlban pótolhatjuk. Íme a hiányzó sor:

```
AddType application/x-httpd-php
➥.php3 .php
```

Ha szeretnénk tudni hogy hogy állnak a dolgok, töltsük fel ezt az egyszerű kis kódrészletet a webhelyünkre.

```
<HTML>
<HEAD>
<TITLE>PHP beállítások
➥ellenőrzése</TITLE>
</HEAD>
<BODY><? php phpinfo() ?>
</BODY>
</HTML>
```

Kis szerencsével az eredmény megegyezik azzal, amit az *1. ábrán* láthatunk. A `phpinfo()` függvény kiírja, hogy milyen összetevőkkel lett a *PHP* értelmező lefordítva, és mellé az egyes beállítási változók értékét. Ez a függvény nagyon sokat segíthet. A kimenete valószínűleg az első dolog lesz, amire rákérdeznek, ha bármikor segítséget kérünk egy on-line *PHP* fórumon.

2. Írassuk ki a parancsfájlok és a PHP futása során fellépő hibákat

A hibakeresés felgyorsítása érdekében a *PHP*-nak és az Apache webkiszolgálónak is megadhatjuk, hogy hová és miként kell a hibaüzeneteket kiíratni. A *php.ini* fájlban található `error_reporting` változó igazából jelzőbitek sorozata. Mindegyik bit a többitől függetlenül beállítható hogy mutasson-e bizonyos hibákat, vagy sem. Egy ilyen utasítás a példa kedvéért:

```
error_reporting = E_ALL
```

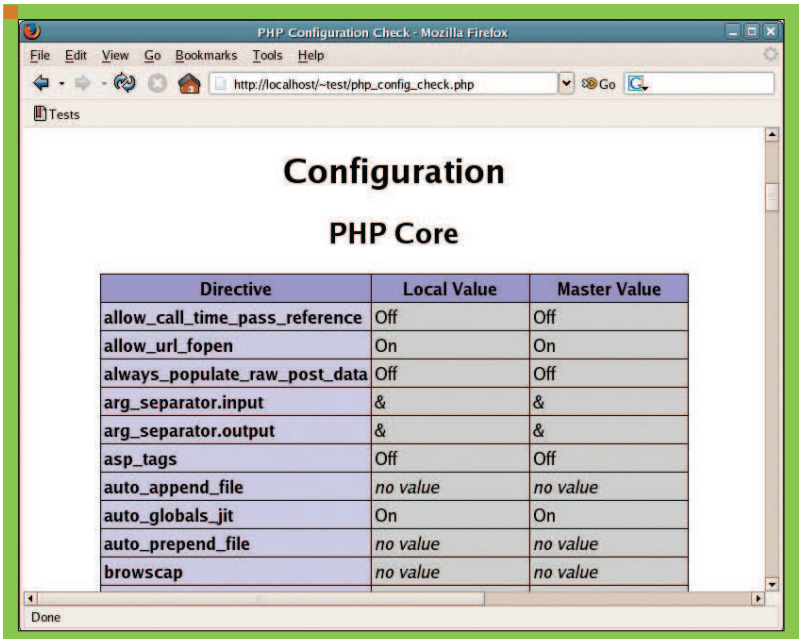
Ennek hatására a környezetünk mindent hibát jelent az egyszerű figyelmeztetéstől a komoly problémákig, de csak abban az esetben, ha a `display_errors` utasítás be van kapcsolva. A *php.ini*-ben megadott általános beállítások felülbírálhatók a webkiszolgáló szintjén is.
Ha *Apache*-ot használunk, a fenti utasítással egyenértékű, ha azt mondjuk a *httpd.conf*-ban, hogy:

```
php_flag  display_errors     on
php_value error_reporting  2047
```

Igazából a *PHP*/webkiszolgáló beállításaihoz nem kellene, hogy legyen hozzáférésünk, és ez gyakran így is van. Ilyen esetben ugyanazt a hatást érhetjük el, ha parancsfájlunkba beszúrjuk az alábbi sort:

```
error_reporting(E_ALL);
```

Ha már a webkiszolgálóknál tartunk: a kiszolgáló hibája esetén ne felejtsük el ellenőrizni a hibanaplóikat, hogy



1. ábra A phpinfo() függvény által kiírt teljes PHP információ

pontosan tudjuk melyik sorban lévő utasítás vezetett az összeomláshoz. Ha a fenti trükkök során már nem találunk hibát, de a programunk még mindig nem fut le helyesen, akkor valószínűleg magában az algoritmus logikájában van a hiba. Valahol valamelyik változó kap egy olyan értéket, ami szerintünk nem is lehetséges, és a kód további részében ez okoz galibát. Ugyanez történik, ha ez a változó éppenséggel egy *SQL* utasítás, amit a program futása során raktunk össze és adtunk át az adatbázis kiszolgálónak.
A megoldás, ha kiíratjuk a változó értékét a böngészőnkbe. Ez egyszerűen megoldható a `print()` utasítással, amit általában a *HTML* tartalom előállítására, kiírására használunk. A `die()` parancs ugyanezt teszi, de azonnal megállítja a program futását is.

3. Fejlécek mindenek előtt!

A *PHP* lehetőséget biztosít *HTTP* fejlécek létrehozásához és az ügyfélhez továbbításához, azelőtt hogy a weblapunk elérné a kimenetet. Nem árt azonban tudni, hogy a `header()` függvényt az összes *HTML* kód és *PHP* kimenet előtt kell meghívni, ideértve az üres sorokat és sortöréseket is! Az alábbi kód például nem fog működni:

```
<?php /* bármilyen PHP
➥kódrészlet lehet itt*/ ?>
<?php header("Content-type:
➥image/png"); ?>
```

Az a picike soremelés, ami a két különálló *PHP* kódrészlet között van, azt eredményezi, hogy a *PHP* a szabványos fejléceket küldi ki a böngészőnek, ami szinte mindig más, mint amit mi küldeni szerettünk volna (különben nem használtuk volna a `header()` függvényt). Figyeljünk oda, hogy ez a soremelés akár egy másik fájlban is lehet. Éppen ez történik, hogyha egy olyan külső fájlt hívunk be (`include`), ami nem pontosan a ?> karaktersorozattal van lezárva (például soremelés van a fájl végén).
Ez elég gyakori oka a programozók fejfájásának sütiket használó web oldalak készítése során. Az egyetlen módja, hogy a sütik működjenek, ha azelőtt kezeljük le azokat, mielőtt a programunk kiküldi a *HTTP* fejléceket. Ha nem jut eszünkbe, hogy egy sima soremelés már elküldi a fejléceket órákig bámulhatjuk a kódot, azon tűnődve, hogy miért nem működnek ezek a vacak sütik. Végtére is még azelőtt beállítjuk a sütiket, hogy utasítanánk a fejlécek kiküldésére. Kár, hogy figyelmen kívül hagytuk azt a lényegtelen tényt, hogy soremelés van a programunkban (vagy egy behívott fájlban), ami a tudtunkon kívül a fejlécek elküldését eredményezi, és ez az, amiért nem működnek a sütik.

4. Mindig ellenőrizzük a felhasználótól jövő adatokat (és vigyázzunk az e-mail címekkel)

Minden esetben jóvá kell hagynunk az oldalunkon beküldött adatokat. A *JavaScript* kódok, amik a böngészőbe írt adatokat ellenőrzik, biztonsági szempontból értéktelenek. Ettől még semmi sem akadályozza meg betörőket, hogy kártékony adatokat küldjön közvetlenül a programkódunknak. Képzeljünk el egy *PHP* bevásárlókosarat, ami csak azokat a termékeket mutatja, ami felhasználó által megadott `$LEGMAGASABB_AR` változó értékénél olcsóbb. Ha mindenféle előzetes ellenőrzés nélkül vidáman lefuttatunk egy *SQL* utasítást, ami a `$LEGMAGASABB_AR` változóban található, és aminek az értéke az általunk hitt helyett valami olyasmi, hogy 'törölj mindent az adatbázisomból', akkor ne lepődjünk meg, ha egyszer csak üresen találjuk a webboltunkat.
Három módszer együttes alkalmazásával ellenőrizhetjük az adatokat.
Az első az adatok szabályos kifejezésekkel történő ellenőrzése, amely félreérthetetlenül meghatározza, hogy milyen az adatok megengedett alakja; egy telefonszám, egy születési év, csak számjegyeket tartalmazhat, szóval átereszthetjük az `is_digit()` függvényen.
A második módszer az `EscapeShellCmd()` függvényhez hasonló lehetőségek használata, amely megakadályozza, hogy a beérkező adatok nemkívánatos parancsokat indíthassanak el a rendszeren, vagy a `mysql_escape_string()` függvény használata olyan változókon, amit *SQL* utasításokba illesztünk be.
A harmadik módszer szorosan összefügg az ellenőrizendő változó adott környezetben vett jelentésével. Itt csak magunkra hagyatkozhatunk. Például az 5555 csak számokból áll, de mégsem valódi telefonszám. Csak akkor szabad elfogadni, ha olyan országból webezik a felhasználónk, ahol létezik ilyen alakú telefonszám. Hasonlóan a 18 egy tökéletesen helyes `$ELETKOR`,

de a parancsfájlunk, amely árengedményt kínál az időseknek vissza kell hogy utasítsa, nemde?
Az e-mail címek ilyen szempontból különösen kellemetlenek. Számtalan függvény van, ami egy e-mail cím szintaktikai helyességét ellenőrzi, mint például ez is a ➲ *www.zend.com/tips/tips.php?id=224&single=1* oldalon. Nem garantálja azonban, hogy ez a cím valóban ahhoz a felhasználóhoz tartozik, aki beküldte, vagy hogy egyáltalán létezik-e mondjuk a *Luke.Skywalker@whitehouse.gov* cím. Mi több igen valószínű , hogy egyáltalán nem is dolgozik *Luke Skywalker* a *Fehér Házban*. Mindig kérjünk a felhasználótól megerősítő e-mailt, vagy nyissunk egy foglalatot a kiszolgálón, hogy egyáltalán létezik-e azon ilyen cím.

5. Használjuk helyesen az idézőjeleket és a bevédett (escape) karaktereket

Mi fog látszani a böngészőnkben, ha lefuttatjuk ezt a nagyon egyszerű *PHP* kódot?

```
<? php
$HOME = 'egy jó hely';
print "1: $HOME<br>"; // dupla
➥idézőjelek
print '2: $HOME<br>'; // egyes
➥idézőjelek
?>
```

A válasz a következő két sorban található:

```
1: egy jó hely
2: $HOME
```

A dupla idézőjeleken belüli változókat kicseréli a *PHP* azok aktuális értékére. Az aposztrófokra viszont úgy tekint, mint egy egy darabból álló tudat nélküli blokkra, amit nem kell módosítani, csak kimásolni a kimenetre. Ugyanez történik, amikor a társításos tömbök kulcsainak elkészítéséhez használunk idézőjeleket.
A `$sajat_tomb['$HOME']` és a `$sajat_tomb["$HOME"]` különböző elemekre mutat. Ez ilyen egyszerű. Még így is nagyon könnyű elfelejteni ezt a megkülönböztetést, és a másikat használni az egyik helyett, vagy épp teljesen megfeledkezni a használatukról. Így hát ha valaminek nem az az értéke, amit vártunk, először ellenőrizzük, hogy milyen idézőjeleket használtunk.
Mivel a felhasználó által küldött adatokban nem bízhatunk, beállíthatjuk a *PHP*-t, hogy védje be (escape) az összes különleges karaktert, ami a *HTML* űrlapból beküldött `$_POST` tömbben megtalálható. Így azonban még a *PHP*-ra vonatkozó belső adatok is tartalmazhatnak bevédett karaktereket, amiket vissza kell állítanunk, mielőtt feldolgoznánk azokat. Erre való a `stripslashes` függvény, amit az alábbi, on-line *PHP* kézikönyvből származó példa mutat be:

```
<?php
$str = "A te neved O\'reilly?";
// Kimenet: A te neved
➥O'reilly?
echo stripslashes($str);
?>
```

6. Hagyjuk, hogy az adatbázis dolgozzon a parancsfájlunk helyett

Ahogy már szóltunk róla, a *PHP*-t olyan gyakran használják a *MySQL*-lel együtt, hogy a *LAMP* betűszó egyike a leginkább ismert fogalmaknak a webfejlesztés területén. Következésképpen a gyorsabb *PHP* parancsfájlok készítésének egyik legjobb módja, ha kellőképp megtanuljuk a *MySQL* használatát, hogy amennyit csak lehet, dolgozzon az a *PHP* helyett.
A következő kódrészlet jól szemlélteti a fenti elgondolást:

```
<?php //keressük meg az összes
➥könyvet, amit Asimov írt 1980
➥után
$sql = "select YEAR, BOOK from
➥MY_BOOKSHELF where AUTHOR
➥LIKE 'Asimov' ; ";
  if ($sql_res = mysql_query
➥("$sql")) {
    while ($r  = mysql_fetch_
➥array($sql_res)) {
      if ($r[YEAR] > 1980) {//
➥írjuk ki a könyv címét ;}
   }
?>
```

És:

```
<?php //keressük meg MySql
➥segítségével az összes
➥könyvet, amit Asimov írt 1980
➥után
$sql = "select BOOK from
➥MY_BOOKSHELF where AUTHOR
➥LIKE 'Asimov'  AND YEAR >
➥1980;";
  if ($sql_res = mysql_query
➥("$sql")) {
    while ($r = mysql_fetch_
➥array($sql_res)) {
      // csak írjuk ki az
➥összes visszakapott címet ;
}
?>
```

A második változat sokkal gyorsabban fog futni, mint az első, mivel az adatbázismotorokat arra tervezték, hogy kiválasszák a különböző feltételeknek megfelelő adatokat olyan gyorsan, amennyire csak lehetséges. Az ilyen feladatok esetében az adatbázis mindig gyorsabb lesz, mint a *PHP*. Győződjünk meg tehát arról, hogy az összes lehetséges kiválasztási logikát *SQL* lekérdezéseken belül helyeztük el, és nem az azt összeállító és futtató *PHP*-ben. Ez a tipp természetesen mindegyik adatbázismotorra vonatkozik, amit a *PHP*-vel használhatunk.

7. Készítsünk hordozható fájlkezelő kódot

A szöveges fájlokban a sorvégeket különböző módon értelmezzük az egyes operációs rendszereken. A bináris fájlok, mint például a képek vagy a tömörített állományok sokkal rosszabbak abban az értelemben, hogy egyetlen rossz karakter a teljes állományt használhatatlanná teszi. Innentől kezdve csak tőlünk függ, hogy olyan kódot akarunk-e írni, ami minden rendszeren helyesen kezeli a fájlokat. Ez arra az esetre is érvényes, ha biztosak vagyunk benne, hogy az ügyfélprogramunk és a webkiszolgálónk is *GNU/Linux* környezet alatt működik. Ha nem így döntünk egészen addig nem fogunk hibát találni a képfájlban, vagy a szövegfeldolgozó kód működése során, amíg egyszer az egyik ismerősünk Windows vagy Apple számítógépéről töltünk fel egy fájlt. Ami a *PHP*-t illeti, a megoldás a `fopen()` függvény `t` (szöveges módú értelmezés (***text mode translation***) és `b` (***bináris***) jelzőbitjeinek helyes használata. A véres részleteket megtaláljuk a ➲ *www.php.net/function.fopen* oldalon. Ne feledjük, hogy az oldal

kifejezetten ezt javasolja: „a hordozhatóság érdekében nagyon ajánlott a t jelzőbitet használó vagy arra építő kódok átírása”.

8. Ismerjük a szövegfeldolgozó függvényeket

A web oldalak még mindig főként szöveges elemekből állnak össze, ami ugyancsak igaz a legtöbb adatbázisra. Ezért van az, hogy a szövegelemzések, szövegfeldolgozások hatékonyabbá tétele a legkönnyebb módja, hogy gyorsabbá tegyük a parancsfájljainkat A szabályos kifejezések rendeltetése valami ilyesmi volna, de olyan, mintha hieroglifákat olvasnánk, és nem is biztos, hogy a legjobb megoldást nyújtja. A *PHP*, habár nem megy annyira a dolgok mélyére (gondolok itt a *Perl* erejére és rugalmasságára), nem is egy olyan függvénye van, ami a szabályos kifejezésekhez hasonlóan működik, csak sokkal gyorsabban. Gondolok itt az `str_replace()`, `strcmp()`, `strtolower()`, `strtoupper()`, `strtr()`, `substr()`, `trim()`, `ucfirst()` és még számos más függvényre. Szánjunk rá időt, hogy tanulmányozzuk őket a kézikönyvben, megéri.

9. Válasszuk szét a kódot és a kinézetet

Biztos módja, hogy olvashatatlanná és nehezen javíthatóvá tegyük a web oldalunk kódját, ha összefésüljük a nagy darab *PHP* kódrészleteket és a *HTML* kódot, még akkor is, ha mindegyik *PHP* fájl csak egyetlen egyszer szerepel az adott oldalban, mint az alábbi példánkban is:

```
myfile.php>
<!– egy csomó HTML kód az
➥állandó fejléchez, menü,
➥logó-k...>
<?php egy csomó PHP kód, amely
➥a legfrissebb hírek listáját
➥készíti el?>
<!– egy csomó HTML kód az oldal
➥középső részéhez...>
<?php egy csomó PHP kód, amely
➥felhasználónként elkészíti
➥a legnépszerűbb oldalak
➥listáját ?>
<!– egy csomó HTML kód
➥a felhasználói visszajelzések
➥űrlap számára...>
```

Ahelyett, hogy elkövetnénk ezt a hibát, foglaljunk minden *PHP* kódrészletet egy vagy több függvénybe, majd tegyük mindegyiket különálló fájlokba (*HTML* kód nélkül), amit aztán az `include_once()` paranccsal hívhatunk be. Az eredmény sokkal tisztább, és egyszerűbben karbantartható:

```
myfile.php>
<?php include_once
➥("common_code.php"); ?>
<!- egy csomó HTML kód az
➥állandó fejléchez, menü,
➥logó-k...>
<?php legutobbi_hirek (); /*
➥csak a függvényhívás */ ?>
<!- egy csomó HTML kód az oldal
➥középső részéhez...>
<?php legnepszerubb_oldalak ();
➥/* csak a függvényhívás */ ?>
<!- egy csomó HTML kód a
➥felhasználói visszajelzések
➥űrlap számára...>
```

Másik nagy előnye ennek a megközelítésnek, hogy a *common_code.php* egyszerű behívásával (ahogy az a fenti példában is történt) a webalkalmazásunk bármelyik oldala használhatja ugyanezeket a függvényeket. Még ennél is fontosabb, hogy a módosítandó függvények új változata azonnal elérhető az összes lap számára.

10. Ellenőrizzük a függvényhívások és a rendszerhívások eredményeit

Végezetül, de nem utolsósorban: minden *PHP* funkciónak a hívó felé elfogadható eredményt kell visszaadnia! A trükkös része ennek a nyilvánvalóan felesleges megállapításnak az a tény, hogy az elfogadható eredmény jelentése a teljes parancsfájltól függ, és minden pillanatban különböző lehet. Itt egy nagyon buta, de hatékony példa, hogy mire gondolok:

```
function kivonas($A, $B) {
        $kulonbseg = $A - $B;
        return($kulonbseg);
}
   $C = 1/kivonas(3, 3); // HI-
➥BA! Nullával való osztás!
   $D = 1/(1 - kivonas
➥(3,3);
```

Habár `$C` kiszámítása a parancsfájl futásának összeomlásához vezet, `$D` kiszámítása ugyanazokkal a kivonandó számokkal lefut. A lényeg az, hogy mielőtt bármit is kezdenénk egy változóval, ellenőrizzük, hogy az értéke elfogadható-e. A fenti példában ez azt jelenti, hogy a `kivonas()` függvény eredményét egy segédváltozóba kellett volna tenni, és csak akkor folytatni az osztással, ha az értéke nem nulla. Még fontosabb, hogy leellenőrizzük a rendszerhívások visszatérési értékét. A rendszerhívások itt olyan függvények, amik arra valók, hogy külső folyamatokkal és fájlokkal végezzünk műveleteket. Nem szabad figyelmen kívül hagyni egyetlen visszatérési értéket sem, különben úgy történhet adatvesztés, hogy nem is vesszük észre. Lásd az alábbi példát:

```
<?php
$HANDLE = fopen
➥("ujfelhasznalo.txt","w"));
➥// megnyitjuk a fájlt
fwrite($HANDLE, "Az új
➥felhasználó adatai");
➥// beleírjuk
?>
```

Ha a `fopen()` meghiúsul (például mert megtelt a lemez, vagy nincs jogosultságunk az íráshoz), az új felhasználó adatai el fognak veszni. Az írás művelet előtt ellenőrizzük a `$HANDLE` értékét, hogy null-e:

```
<?php
if (!$HANDLE = fopen
➥("ujfelhasznalo.txt","w"))
➥{ die "Hozzáférés sikertelen:
ujfelhasznalo.txt"; }
fwrite($HANDLE, "Az új
➥felhasználó adatai");
?>
```

Kellemes PHP programozást!

Linux Journal 2006., 145. szám

Marco Fioretti főként hardverrel foglalkozó rendszermérnök, aki a szabad szoftverek világában is érdekelt. Egyrészt az EDA felülettel kapcsolatosan, másfelől ő a hatékony munkaállomásokat megcélzó RULE projekt vezető fejlesztője. Marco a családjával együtt Rómában él.