KAPITOLA 3

Objekty a OOP

V této kapitole:

- První kroky s OOP
- Rozšiřujeme třídy
- Šablony
- Soubory projektu

V této knize budeme používat objektově **orientované programování** (zkráceně **OOP**). Jedná se o styl programování, v němž vytváříme objekty, které obsahují data i funkce, jež s těmito daty pracují. Protože objekty sdružují data a funkce na jediném místě, zachováváme uspořádanost zdrojového kódu a lépe zvládáme složitost, jak naše aplikace roste.

Objekty vytváříme pomocí **tříd**. Vztah mezi třídami a objekty lze snadněji pochopit na příkladu stavaře. Stavitel (v tomto případě jazyk PHP) se řídí plánem (třídou), aby postavil dům (objekt).

Stejně jako stavař může postavit více domů podle jediného plánu, jazyk PHP může vytvořit více instancí objektu. Každý z domů bude mít stejné dispozice, ale pravděpodobně bude jinak vymalovaný a bude v něm žít jiná rodina. V jazyce PHP má každý objekt vytvořený z jedné třídy stejnou funkčnost, ale vlastnosti a data, která „přebývají“ v objektech, budou jedinečná pro každou instanci.

OOP se může zdát na první pohled složité, ale jedná se o velmi jednoduchý styl programování. Více informací o OOP lze získat z videa Lorny Mitchellové na stránkách sitepoint.com.[18]

První kroky s OOP

Napišme krátký objektově orientovaný kód společně. Předpokládejme, že budeme chtít reprezentovat psa jako objekt v našem kódu jazyka PHP. Musíme napsat definici pro tento objekt (třídu) a v ní popíšeme, co může pes dělat.

18 *http://www.sitepoint.com/object-oriented-php-lesson-1/*

Vytvořte nový soubor *pes.php* v adresáři *muj_projekt* a vložte do něho následující kód:

```
<?php
class Pes {
  public $jmeno;

  public function __construct($jmeno) {
    $this->jmeno = $jmeno;
  }

  public function mluv() {
    return 'Haf! Haf!';
  }
}
```

Nyní vytvořte v adresáři *muj_projekt* další soubor s názvem *pes_test.php* a s tímto obsahem:

```
<?php
require 'pes.php';

$pes = new Pes('Fido');
echo 'Pes se jmenuje: ' . $pes->jmeno . '<br />';
echo 'Pes říká: ' . $pes->mluv() . '<br />';
```

Otevřete skript *pes_test.php* ve svém webovém prohlížeči a měli byste uvidět jméno psa, a také, jak tento pes štěká.

Poznámka: Vkládání kódu z dalších souborů

Příkaz `require` z výše uvedeného příkladu (jednoduše řečeno) importuje obsah externího souboru s kódem jazyka PHP do našeho souboru *pes_test.php*. Jedná se o jeden ze čtyř příkazů pro vkládání externího kódu, přičemž každý z nich se chová mírně odlišně:

- `include` – vkládá obsah souboru. Pokud jazyk PHP nenajde tento soubor, nebo z něho nemůže číst, vypíše varování, ale pokračuje dále v provádění skriptu.
- `include_once` – funguje podobně jako příkaz `include`, ale jazyk PHP si ověřuje, jestli už stejný soubor nenaimportoval. Jestliže ho už naimportoval, nevloží ho znovu.
- `require` – téměř stejný jako příkaz `include`, ale zastaví provádění skriptu a vypíše fatální chybu, když nenajde požadovaný soubor.
- `require_once` – podobá se příkazu `require`, ale navíc kontroluje, jestli už nevkládá stejný soubor podruhé.

Když si prohlédnete soubor *pes.php*, na první pohled si všimnete klíčového slova `class`. Toto klíčové slovo říká, že za ním následuje kód pro definici třídy, která bude známá pod názvem, jenž bezprostředně následuje za tímto klíčovým slovem (v tomto případě se jedná o název `Pes`). Jakmile definujete tuto třídu, můžete z ní vytvořit objekt pomocí konstrukce `new Pes`.

Metody `__construct()` a `mluv()` jsou součástí definice naší třídy. Metoda `__construct()` má pro jazyk PHP zvláštní význam – když vytváříme nový objekt z třídy, jazyk PHP v ní zkouší hledat právě tuto metodu. Pokud ji najde, automaticky ji spustí hned poté, co vytvoří instanci nového objektu. Tato metoda je rovněž známá pod pojmem **konstruktor**. Jedná se o skvělé místo, kam umístit kód pro inicializaci nového objektu.

Poznámka: Funkce a metody

Funkce náležící třídě se nazývají **metody**. Mezi funkcemi a metodami je jen minimální rozdíl, takže je v pořádku, když si představíte „funkci" pokaždé, když uvidíte slovo „metoda", pokud vám to pomůže ke snadnějšímu pochopení.

Třída může mít svou vlastní sadu proměnných, které pomáhají objektu udržet si stav, a její metody k nim můžou přistupovat. Na řádku `public $jmeno;` našeho souboru *pes.php* definujeme `$jmeno` jako proměnnou třídy (jinak řečeno **vlastnost**).

Poznámka: Proměnné a vlastnosti

Proměnné, které náleží třídě, bývají označovány jako **vlastnosti**. Podobně jako u funkcí a metod – pokud se lépe sžijete s terminologií OOP tak, že si představíte „proměnnou", kdykoli uvidíte slovo „vlastnost", směle do toho.

Když píšeme kód uvnitř třídy, můžeme používat speciální proměnnou `$this`, která představuje aktuální instanci třídy (objekt) a pomáhá nám správně přistupovat k metodám a vlastnostem. Pro ukázku zkusme upravit soubor *pes_test.php*, aby vypadal takto:

```
<?php
require 'pes.php';

$fido = new Pes('Fido');
echo 'Pes se jmenuje: ' . $fido->jmeno . '<br />';
echo 'Pes říká: ' . $fido->mluv() . '<br />';

$fifinka = new Pes('Fifinka');
echo 'Pes se jmenuje: ' . $fifinka->jmeno . '<br />';
echo 'Pes říká: ' . $fifinka->mluv() . '<br />';
```

Přejmenovali jsme proměnnou `$pes` na `$fido` a vytvořili jsme druhou instanci třídy `Pes`, kterou jsme uložili do proměnné `$fifinka`. Po spuštění tohoto kódu by se měli na obrazovce objevit dva štěkající psi Fido a Fifinka. Co se ale odehrává na pozadí?

Oba tyto objekty pocházejí ze stejné třídy (`Pes`), ale každý z nich reprezentuje samostatnou instanci s vlastními daty – například každý z nich má svou vlastnost `$jmeno`. Při vytváření instance `$fido` ukládáme do proměnné `$jmeno` textový řetězec `'Fido'`, kdežto u instance `$fifinka` do ní ukládáme textový řetězec `'Fifinka'`.

Tip: Hlídejte si šipky

Když voláte metodu nebo přistupujete k vlastnosti, nezapomeňte na operátor `->`. Tím dáváte jazyku PHP najevo, že se odkazujete na něco, co patří třídě/objektu, a ne na běžnou proměnnou nebo funkci. Proto na to nezapomínejte při volání metod a přístupu k vlastnostem, jinak se můžou vaše skripty začít chovat neočekávaně.

Rozšiřujeme třídy

Nyní známe základy OOP, takže si je stručně zopakujme. Třída je definice, která seskupuje proměnné a funkce do logického celku. Konkrétní instanci, kterou vytváříme na základě třídy, nazýváme objekt. OOP samozřejmě nepojednává jen o objektech a třídách. Dalším důležitým aspektem je schopnost rozšiřovat třídu, abychom do ní mohli přidávat funkčnost (nebo ji zlepšovat), aniž bychom museli znovu programovat funkce, které se nemění.

Tato koncepce se nazývá **dědičnost** – tvorba nové třídy tak, že rozšiřujeme stávající třídu. Vytvořme nový soubor *mazlicek.php* s definicí třídy `Mazlicek`. Ta bude sloužit jako základní třída, kterou budeme rozšiřovat dalším třídami pro konkrétní mazlíčky – například psa, kočku, rybu, ještěrku atd.

```
<?php
class Mazlicek {
  public $jmeno;

  public function __construct($jmeno) {
    $this->jmeno = $jmeno;
  }

  public function mluv() {
    return 'nic';
  }
}
```

Tato třída se záměrně podobá naší třídě `Pes`, takže by na ní nemělo být nic překvapujícího. Přepišme ale třídu `Pes`, aby rozšiřovala třídu `Mazlicek` a používala její funkčnost.

```
<?php
require_once 'mazlicek.php';

class Pes extends Mazlicek {
  public function mluv() {
    return 'Haf! Haf!';
  }

```

```
  public function hraj() {
    return 'přines';
  }
}
```

Nyní přidáme několik dalších zvířecích tříd. Například do nového souboru *kocka.php*:

```
<?php
require_once 'mazlicek.php';

class Kocka extends Mazlicek {
  public function mluv() {
    return 'Mňau!';
  }

  public function hraj() {
    return 'chyť myš';
  }
}
```

a potom do souboru *ryba.php*:

```
<?php
require_once 'mazlicek.php';

class Ryba extends Mazlicek {
}
```

Teď si je vyzkoušíme. Vytvoříme soubor *mazlicek_test.php* a vložíme do něho níže uvedený zdrojový kód:

```
<?php
require 'pes.php';
require 'kocka.php';
require 'ryba.php';

$fido = new Pes('Fido');
echo 'Pes se jmenuje: ' . $fido->jmeno . '<br />';
echo 'Pes říká: ' . $fido->mluv() . '<br />';
echo 'Pes si hraje: ' . $fido->hraj() . '<br />';

$berta = new Kocka('Berta');
echo 'Kočka se jmenuje: ' . $berta->jmeno . '<br />';
echo 'Kočka říká: ' . $berta->mluv() . '<br />';
echo 'Kočka si hraje: ' . $berta->hraj() . '<br />';
```

```
$nemo = new Ryba('Nemo');
echo 'Ryba se jmenuje: ' . $nemo->jmeno . '<br />';
echo 'Ryba říká: ' . $nemo->mluv() . '<br />';
```

Nyní to začíná být zajímavé. Zkuste vytvořit třídy pro papouška a ještěrku a následně vytvořte jejich instance v souboru *mazlicek_test.php*. S jejich pomocí zobrazte jejich jména a co tak přibližně můžou říkat.

Když rozšiřujeme třídu, tak ve skutečnosti vytváříme novou třídu z jiné. Tímto způsobem vzniká vztah mezi rodičovskou a dceřinou třídou. Například bychom mohli prohlásit, že třída `Pes` je dceřinou třídou třídy `Mazlicek` a že třída `Mazlicek` je rodičovskou třídou pro třídu `Pes`. Když stavíme na již napsaném kódu, nemusíme psát stejný kód na různých místech – díky tomu získáme lépe uspořádaný a efektivnější kód.

Každá třída, která reprezentuje určitý typ mazlíčka, získává své metody a vlastnosti z rodičovské třídy `Mazlicek`. Vezměme si kupříkladu třídu `Ryba`, jež nemá žádný vlastní kód, ale veškeré chování zdědila od třídy `Mazlicek`.

Na druhou stranu – třídy `Pes` a `Kocka` rozšiřují třídu `Mazlicek`. Kromě toho přepisují metody `speak()` svými vlastními verzemi a doplňují vlastní funkčnost v podobě metody `hraj()`.

Poznámka: A co klíčové slovo public?

V třídách připojujeme k našim vlastnostem a metodám **viditelnost**. Když přidělujeme viditelnost, můžeme si vybrat některé ze tří klíčových slov: `public`, `protected` nebo `private`. Klíčové slovo `public` (veřejné) označuje vlastnosti a metody, k nimž lze přistupovat odkudkoli. Klíčové slovo `protected` (chráněné) mají vlastnosti a metody, ke kterým můžou přistupovat pouze metody ze stejného stromu tříd (ve smyslu stromu dědičnosti). Klíčové slovo `private` (soukromé) označuje metody a vlastnosti, ke kterým můžou přistupovat pouze metody ze stejné třídy.

Více informací o viditelnosti uvnitř tříd je k dispozici v následujících článcích:

- *http://php.net/manual/en/language.oop5.visibility.php*
- *http://aperiplus.sourceforge.net/visibility.php*

Právě jsme si popsali základy OOP v jazyce PHP. Přestože se tento styl programování může zdát na první pohled složitý, podobá se jízdě na kole – jakmile se ho naučíme, už ho nezapomeneme.

OOP usnadňuje tvorbu rozsáhlých aplikací a setkáte se s ním v téměř všech profesionálních programovacích jazycích. Pokud se ho naučíte používat v jazyce PHP, jednodušeji v budoucnu pochopíte jiné programovací jazyky, takže se jedná o skutečně užitečnou dovednost.

Zbytek této knihy bude předpokládat, že rozumíte OOP a umíte používat třídy a objekty v jazyce PHP. Pokud se s ním chcete seznámit blíže, prohlédněte si některé z níže uvedených článků:

- *http://programujte.com/clanek/2009113001-oop-v-php/*
- *http://codular.com/introducing-php-classes*
- *http://www.php5-tutorial.com/classes/introduction/*

- *http://php.net/manual/en/keyword.extends.php*
- *http://jadendreamer.wordpress.com/2011/05/13/php-tutorial-learning-oop-class-basics-extending--classes/*
- *http://www.killerphp.com/tutorials/object-oriented-php/*
- *http://www.techotopia.com/index.php/PHP_Object_Oriented_Programming*
- *http://www.techflirt.com/tutorials/oop-in-php/index.html*
- *http://code.tutsplus.com/tutorials/object-oriented-php-for-beginners--net-12762*

Šablony

V kapitole 1, „Server", jsme si řekli, že do souborů PHP můžeme ukládat jak kód jazyka PHP, tak kód jazyka HTML. Díky tomu je jazyk PHP skvělou vstupní bránou do světa programování na straně serveru pro všechny kodéry ovládající jazyk HTML. U složitých aplikací je však kombinování kódu v jazycích HTML a PHP velmi nepřehledné. Větší vývojářské týmy mají navíc programátory specializující se na frontend (prezentaci) a backend (zpracování dat na serveru). Mixování obou těchto kódů je v takovém prostředí nežádoucí, protože jednotliví vývojáři si nechtějí navzájem zasahovat do kódu.

Šablonování nám umožňuje oddělit prezentační logiku od zpracování dat. Existuje spousta vzorů podporujících OOP a šablonování, přičemž nejznámější je architektura MVC (Model-View-Controller), která se objevuje také ve spoustě oblíbených frameworků, kterými jsou kupříkladu CakePHP,[19] Zend Framework[20] a CodeIgniter.[21] Architektura MVC rozděluje kód aplikace na různé oblasti zájmu. Tímto způsobem lépe uspořádává náš zdrojový kód.

Existuje spousta šablonovacích systémů, ale jednoduše můžeme používat i soubor PHP, v němž převažuje kód jazyka HTML. Cílem fragmentů kódu jazyka PHP v takovém souboru je vypisovat jen obsahy proměnných. Proměnné obvykle nastavuje v nějakém jiném souboru, který obsahuje pouze kód jazyka PHP a vkládá tuto šablonu. Ukažme si příklad.

Soubor šablony by vypadal takto:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title><?php echo $nazevStranky; ?></title>
</head>
<body>
  <ul>
```

19 *http://cakephp.org/*

20 *http://framework.zend.com/*

21 *http://ellislab.com/codeigniter*

```
  <?php foreach ($pole as $prvek) {?>
    <li><?php echo $prvek; ?></li>
  <?php } ?>
  </ul>
</body>
</html>
```

Dále potřebujeme ještě skript PHP, ve kterém nastavíme proměnné a vložíme předchozí šablonu:

```
<?php
$nazevStranky = 'Má ukázková šablona';
$pole = array('jedna', 'dvě', 'tři');
require 'cesta/k/šabloně.php';
```

Vložená šablona zdědí oblast platnosti od volajícího souboru, a tudíž má přístup k jeho proměnným, funkcím, třídám atd.

Výhody tohoto způsobu šablonování jsou:

- Uplatníme ho jednoduše. Nepotřebujeme žádnou knihovnu pro zobrazování šablon, kterou bychom museli načítat do naší aplikace.
- Šablony stále můžou zpracovávat kód jazyka PHP. Můžeme v nich procházet pole, volat funkce atd.

Tento přístup má ale také několik nevýhod:

- Vývojáři frontendu, kteří vytvářejí šablony, musejí znát jazyk PHP. Všichni, kdo budou pracovat s těmito šablonami, musejí znát jazyk PHP.
- Nejedná se o skutečné šablonování. Tyto „šablony“ nejsou ve skutečnosti šablonami – jsou to jen další soubory PHP, které rozvrhují data pro zobrazení.

Jako alternativu můžeme zvolit šablonovací knihovny, které nabízejí vlastní syntaxi. Ty se velmi liší od šablon, které jsme si ukázali před chvílí, jelikož k tvorbě šablony nepoužíváme vůbec jazyk PHP. Místo něho používáme speciální syntaxi, jež je jedinečná pro každou z těchto knihoven a obvykle vypadá jako kombinace kódu a čistého textu. Když taková knihovna zobrazuje šablonu, nahrazuje své speciální výrazy daty, které reprezentují.

Tento přístup k šablonování volí kupříkladu i oblíbený publikační systém Expression Engine,[22] v němž všechny šablony obsahují speciální bloky kódu pro zobrazení dat.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
```

22 *http://ellislab.com/expressionengine*

```
  <title>{% nazevStranky %}</title>
</head>
<body>
  <ul>
  {% if pole as prvek %}
    <li>{% prvek %}</li>
  {% /if %}
  </ul>
</body>
</html>
```

Tato metoda šablonování má následující výhody:

- Není nutné se učit jazyk PHP. Vývojáři šablon si vystačí se seznamem speciálních bloků, s nimiž můžou načítat data z našich skriptů PHP, a proto se nemusejí učit jazyk PHP a jeho syntaxi.
- Šablonovací knihovna převádí tyto šablony na standardní soubory HTML. Tyto soubory si tedy mohou ponechat obvyklou příponu *.html*.

Tento přístup má však také nevýhody:

- Neobejdeme se bez knihovny, která zobrazí šablony. Tuto knihovnu třetí strany musíme vložit do naší aplikace.
- Každá knihovna může mít svou vlastní syntaxi a funkčnost. To může mít vliv na to, že budeme muset data dodatečně zpracovávat, nebo může tato syntaxe kolidovat s naším osobním stylem zápisu kódu.

Oba přístupy mají své výhody i nevýhody, proto bychom si měli vyzkoušet co nejvíce variant, abychom zjistili, co se bude hodit nejlépe pro naši aplikaci. Více informací o šablonování lze najít v těchto článcích:

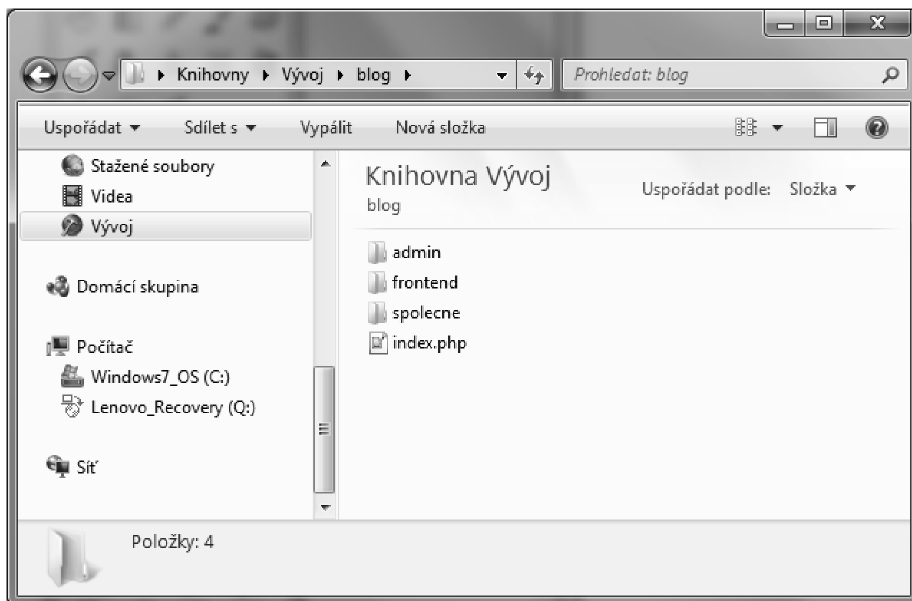
- *http://smarty.ronnieweb.net/co-je-smarty.php*
- *http://www.broculos.net/2008/03/how-to-make-simple-html-template-engine.html#.Uwi4-4XHy8A*
- *http://coding.smashingmagazine.com/2011/10/17/getting-started-with-php-templating/*
- *http://www.sitepoint.com/smarty-php-template-engine/*
- *http://www.sitepoint.com/beyond-template-engine/*

Soubory projektu

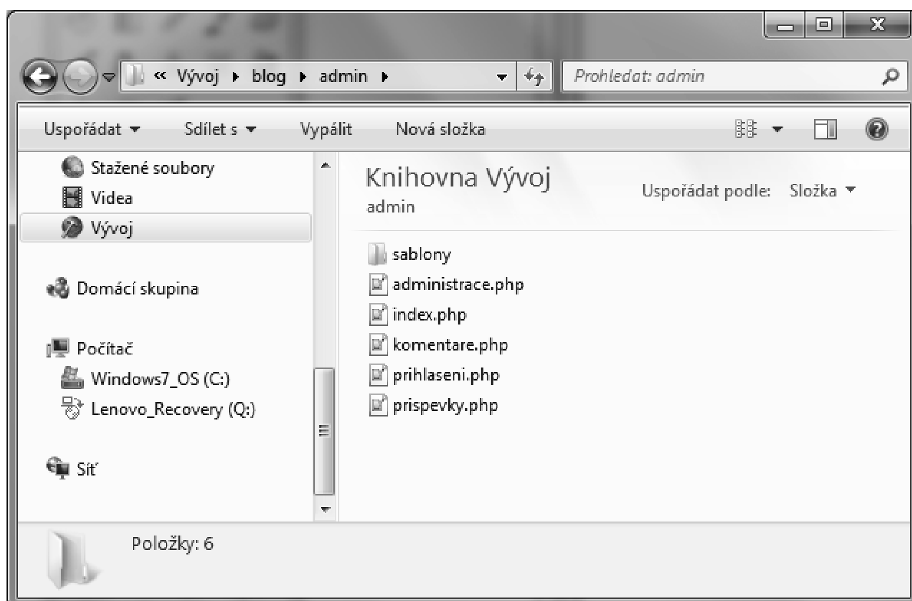
Jelikož jsme si vysvětlili OOP a šablonování, je na čase zapojit tyto nové koncepce do naší blogovací aplikace, díky čemuž si vytvoříme základní framework pro naši aplikaci. Musíme přidat několik souborů:

- obrázek 3.1 ukazuje, jak by měl vypadat kořenový adresář,
- na obrázku 3.2 lze vidět obsah adresáře *admin*,

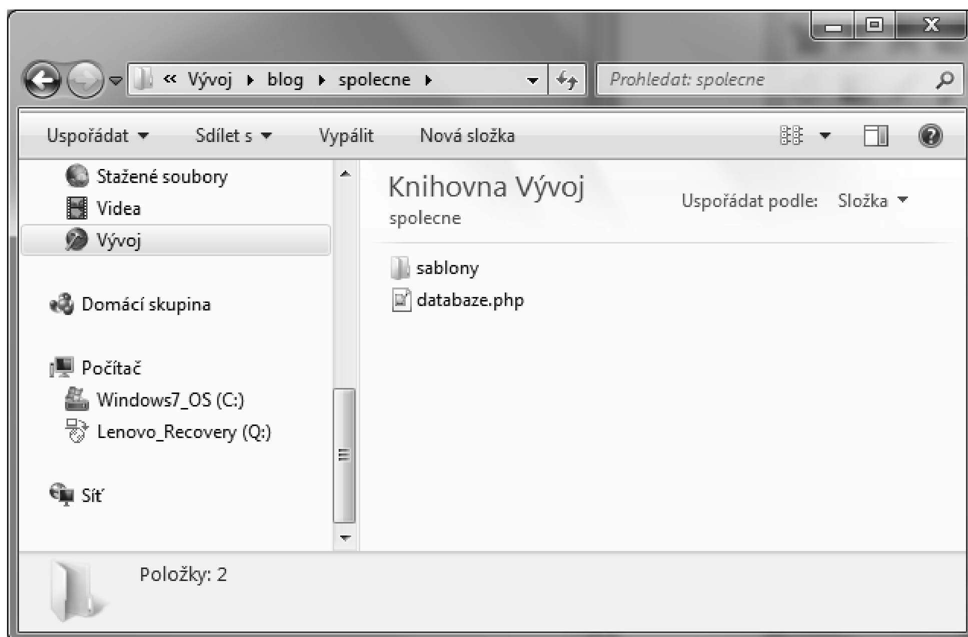
- obrázek 3.3 ukazuje adresář *spolecne*,
- na obrázku 3.4 lze vidět obsah adresáře *frontend*.



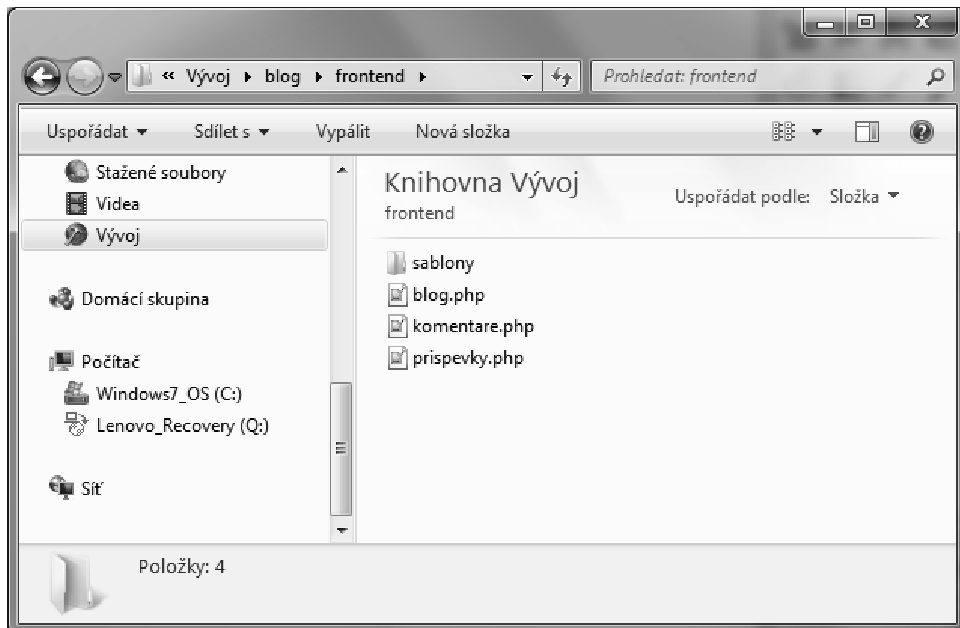
Obrázek 3.1. Kořenový adresář



Obrázek 3.2. Adresář admin



Obrázek 3.3. Adresář spolecne



Obrázek 3.4. Adresář frontend